CI-3 : Implémenter et comparer des méthodes numériques

CI-3-1 Résoudre ou simuler numériquement un problème d'ingénierie

Objectifs — SIMULER-RESOUDRE-VALIDER

A la fin de la séquence, l'élève doit être capable de :

- **C1** Proposer une démarche de résolution
  - Choisir une démarche de résolution d'un problème d'ingénierie numérique ou d'intelligence artificielle.
- **C3** Mettre en œuvre une démarche de résolution numérique
  - Mener une simulation numérique.
    - ⋆ Choisir des grandeurs physiques.
    - ⋆ Choisir du solveur et de ses paramètres (pas de discrétisation et durée de la simulation).
    - ⋆ Choisir des paramètres de classification.
    - ⋆ déterminer l'influence des paramètres du modèle sur les performances.
  - Résoudre numériquement une équation ou un système d'équations.
    - ⋆ Réécrire des équations d'un problème.
    - ⋆ Résoudre un problème du type $f(x) = 0$ (méthodes de dichotomie et de Newton).
    - ⋆ Résoudre un système linéaire du type A.X = B.
    - ⋆ Résoudre un système d'équations différentielles (schéma d'Euler explicite).
    - ⋆ Implémenter une intégration et une dérivation numérique (schéma avant et arrière).

Table des matières

Chapitre 1

Listes et tableaux - quelques outils avec ou sans packages

Sommaire

1.1 Modules

Ce cours permet d'établir des méthodes numériques pour résoudre divers problèmes numériques de Sciences de l'Ingénieur. Néanmoins, des modules existent et permettent de résoudre ces problèmes de façon rapide et optimisée. Ce document détaille les méthodes disponibles dans les modules1 `matplotlib.pyplot`, `numpy` et `scipy` pour résoudre ces problèmes numériques.

Afin de bien définir quelle méthode appartient à quel module, le document utilisera les alias suivants :

```
import math as m
import matplotlib.pyplot as plt
import numpy as np
import scipy as sci
import numpy.linalg as nalg
import scipy.optimize as sciop
import scipy.integrate as scint
```

- `m` pour `math`
- `plt` pour `matplotlib.pyplot`
- `np` pour `numpy`
- `sci` pour `scipy`
- `nalg` pour `numpy.linalg`
- `sciop` pour `scipy.optimize`
- `scint` pour `scipy.scint`

Remarque : il est bien sûr possible d'utiliser `from numpy import *`. Cependant il faut être vigilant ; les deux cas suivants ne conduisent pas au même résultat :

```
from math import *
from numpy import *
```

```
from numpy import *
from math import *
```

Le premier cas permet, par exemple, d'utiliser cos sur un tableau ; le second ne le permet pas (voir 1.2.2).

1.2 Tableaux de floats (avec module)

1.2.1 Répartitions linéaires ou logarithmiques

Objectif : obtenir un tableau avec des valeurs réparties de façons linéaire ou logarithmique.

- `np.linspace(deb, fin, nbp)` avec `deb` la première valeur, `fin` la dernière valeur (incluse) et `nbp` le nombre d'éléments.

```
>>> np.linspace(10, 50, 5)
array([10., 20., 30., 40., 50.])
```

```
>>> np.linspace(0.1, 1.1, 5)
array([0.1 , 0.35, 0.6 , 0.85, 1.1 ])
```

- `np.arange(deb, fin, pas)` avec `deb` la première valeur, `fin` la dernière valeur (exclues) et `pas` le pas entre deux éléments.

```
>>> np.arange(10, 60, 10)
array([10, 20, 30, 40, 50])
```

```
>>> np.arange(0.1, 1.2, 0.25)
array([0.1 , 0.35, 0.6 , 0.85, 1.1 ])
```

- `np.logspace(deb, fin, nbp)` avec `10**deb` la première valeur, `10**fin` la dernière valeur (incluse) et `nbp` le nombre d'éléments.

```
>>> np.logspace(-1, 2, 4)
array([  0.1,   1. ,  10. , 100. ])
```

```
>>> np.logspace(0, 1, 3)
array([ 1.        ,  3.16227766, 10.        ])
```

1.2.2 Vectorisation

Contrairement aux listes (cf 1.3), il est possible d'appliquer des opérations directement au tableau.

```
>>> T = np.linspace(0, np.pi/2, 7)
>>> np.cos(T)
array([1.00000000e+00, 9.65925826e-01, 8.66025404e-01, 7.07106781e-01,
       5.00000000e-01, 2.58819045e-01, 6.12323400e-17])
```

1.2.3 Charger un fichier de points

Le module `numpy` propose la méthode `loadtxt` pour lire un fichier de points. Dans un fichier `.csv`, les colonnes sont séparées par des `;`. Ainsi `donnees = np.loadtxt('donnees.csv', delimiter = ';')` permet d'obtenir un tableau `donnees` avec les valeurs du fichier `donnees.csv`.

Si on connait le nombre de colonnes, il est possible d'associer une variable à chaque colonne avec l'argument optionnel `unpack`. Si les colonnes sont séparées par des tabulations, il faut utiliser `\t` comme valeur de `delimiter`.

```
t, x, v = np.loadtxt('donnees.txt', delimiter = '\t', unpack = True)
```

1.3 Listes de `floats` (sans module)

1.3.1 Construire une liste de valeurs suivant une répartition linéaire

```
deb, fin, nbp = 10, 50, 5
L1 = [deb + i*(fin - deb)/(nbp - 1) for i in range(nbp)]
```

```
deb, fin, pas = 0.1, 1.2, 0.25
v , L2 = deb, []
while v < fin :
    L2.append(v)
    v += pas
```

```
deb, fin, nbp = -1, 2, 4
L5, e = [], deb
for i in range(nbp):
    L5.append(10**e)
    e += (fin - deb) / (nbp - 1)
```

1.3.2 Charger un fichier de points

Tout d'abord, il faut lire les données.

```
fich = open('donnees.csv', 'r')
data = fich.readlines()
fich.close()
```

On obtient une liste des lignes du fichier `donnees.csv`. Reste à en extraire les valeurs.

```
donnees = []
for ligne in data:
    donnees.append(ligne[:-1].split(';'))
```

Remarque : les lignes se terminent par `\n`, c'est pourquoi on élimine ce caractère en utilisant `L[:-1]`.

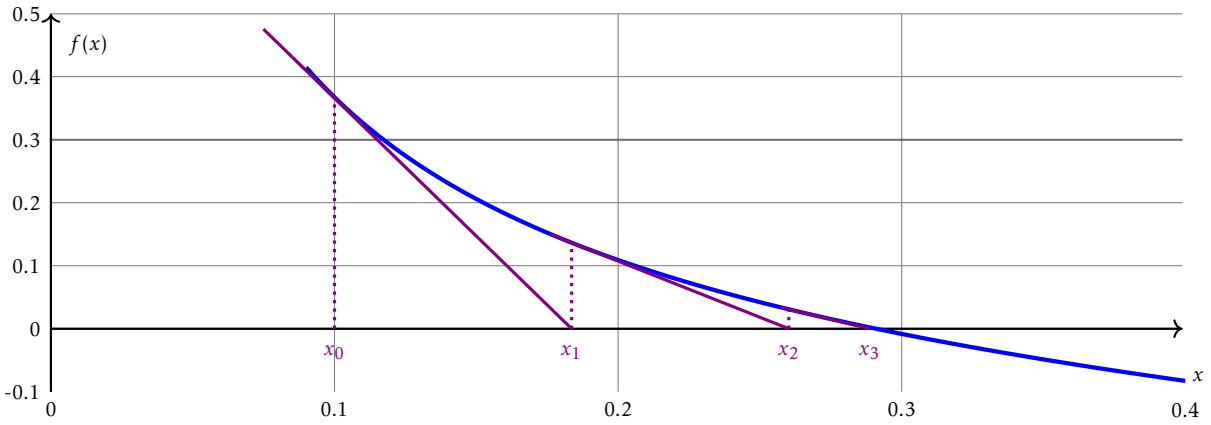
1.3.3 Écrire un fichier de points

On possède une liste `donnees` de listes de valeurs qu'on souhaite ouvrir dans un tableur. Pour cela, on va écrire les données dans un fichier d'extension `.csv`, en séparant les colonnes par des `;`.

```
fich = open('resultats.csv', 'w')
for ligne in donnees:
    a_imp = ''
    for e in ligne:
        a_imp += str(e) + ';'
    fich.write(a_imp[:-1] + '\n')
fich.close()
```

Chapitre 2

Problèmes stationnaires à une dimension



Sommaire

2.1 Introduction

2.1.1 Objectifs

L'objectif de ce cours est de traiter des problèmes stationnaires (constant au cours du temps), linéaires ou non, conduisant à la résolution approchée d'une équation algébrique ou transcendante.

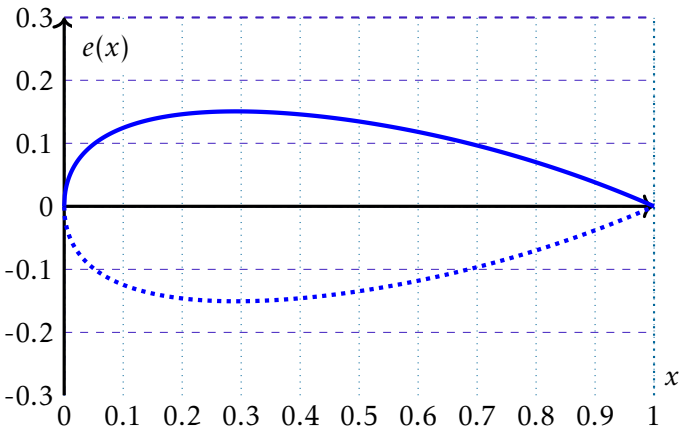
2.1.2 Support de l'étude

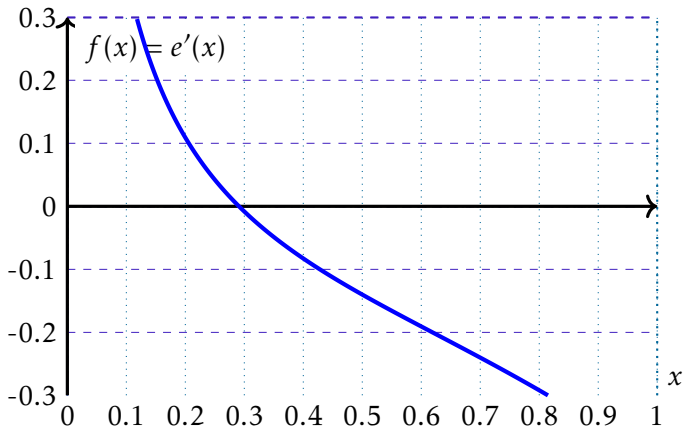
Comme support de cours, nous prenons un profil d'aile d'avion dont la demi-épaisseur $e(x)$ est approchée par l'équation :

$$e(x) = 0,15 \times \left(3,7 \times \sqrt{x} - 3,4 \times x - 0,3 \times x^4\right), \forall x \in [0,1]$$

Rechercher la valeur de x pour laquelle le profil est le plus épais, revient à résoudre l'équation :

$$f(x) = 0,15 \times \left(\frac{3,7}{2.\sqrt{x}} - 3,4 - 1,2 \times x^3\right) = 0$$





2.1.3 Linéarité

Soit f une fonction définie de E, un espace vectoriel, dans K, un corps commutatif. f est linéaire, si elle vérifie la propriété suivante :

$$\forall (x,y) \in \mathrm{E}^2, \forall (\lambda,\mu) \in \mathrm{K}^2, f(\lambda.x + \mu.y) = \lambda.f(x) + \mu.f(y)$$

Exemple :

En 1D : Équation de droite

$$f : \begin{array}{ccc} \mathbb{R} & \to & \mathbb{R} \\ x & \mapsto & a.x + b \end{array} \quad \text{avec} \quad (a,b) \in \mathbb{R}^2$$

En 2D : Définition d'un plan

$$f(x,y,z) = a.x + b.y + c.z + d$$

La forme linéaire $f(x,y,z) = 0$ définit un plan où le vecteur (a,b,c) est un vecteur normal à ce plan.

Résoudre une équation linéaire 1D dans $\mathbb{R}$ ou $\mathbb{C}$ est immédiat :

$$\forall a \neq 0,\ a.x + b = 0 \quad \Rightarrow \quad x = -\frac{b}{a} \qquad \textbf{Exemple :}\ 3.i.x - 12.i + 6 = 0 \quad \Rightarrow \quad x = 4 + 2.i$$

Résoudre un système linéaire de n équations à n inconnues suppose la résolution d'un système matriciel du type :

$$\begin{bmatrix} a_{11} & \dots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \dots & a_{nn} \end{bmatrix} . \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ \vdots \\ b_n \end{bmatrix} \quad \Leftrightarrow \quad [\mathrm{A}].[\mathrm{X}] = [\mathrm{B}] \quad \overset{\det(\mathrm{A}) \neq 0}{\Rightarrow} \quad [\mathrm{X}] = [\mathrm{A}^{-1}].[\mathrm{B}]$$

Nous verrons dans un chapitre suivant comment le résoudre par un pivot de Gauss.

2.1.4 Non linéarité

Les cas non-linéaires sont plus difficiles à résoudre. Parfois une résolution analytique est possible comme dans la résolution d'une équation du second degré :

Exemple :

$$x^2 - x - 6 = 0 \quad \Rightarrow \quad \Delta = (-1)^2 - 4 \times 1 \times (-6) = 25 = 5^2 \quad \Rightarrow \quad \begin{cases} x_+ = \dfrac{-(-1)+5}{2} = 2 \\ x_- = \dfrac{-(-1)-5}{2} = -3 \end{cases}$$

Dans d'autre cas, la solution analytique n'est pas connue (exemple : profil d'aile d'avion). On utilise alors une approche numérique pour la calculer. C'est l'objet de ce cours.

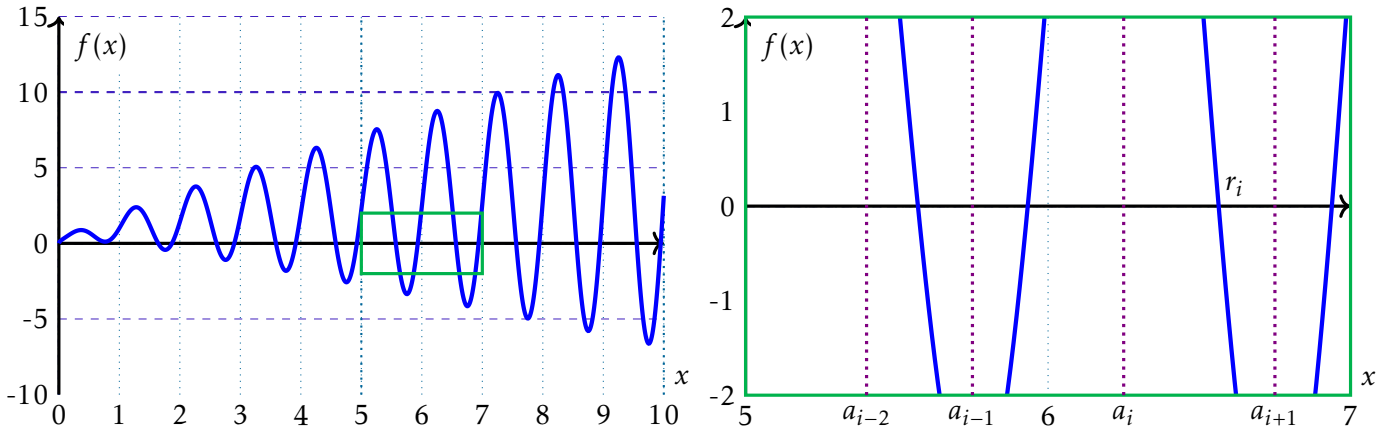
2.2 Convergence

2.2.1 Séparation des racines

La plupart des méthodes de recherche de racines d'une fonction, se comportent bien si une seule racine r (valeur pour laquelle la fonction s'annule) est présente dans l'intervalle d'étude.

Séparer la racines r_i (ième annulation de la fonction sur l'intervalle de départ) revient à trouver un intervalle $]a_i, a_{i+1}[$ où cette racine est unique.

Exemple : $f(x) = \sqrt{x} + x.\sin(2.\pi.x)$



Il convient alors de faire un test aux bornes de l'intervalle d'étude pour éliminer les cas d'inadéquation de la comparaison à zéro, i.e, les cas sans racines sur l'intervalle. Il est en effet, possible d'élaborer une procédure permettant de déterminer la parité du nombre de racines sur l'intervalle :

- Posons $p_i = f(a_i).f(a_{i+1})$
- En tenant compte de l'ordre de multiplicité des racines, nous avons :
  - si $p_i < 0$, il existe $2.n+1$ racines dans $]a_i, a_{i+1}[$, $n \in \mathbb{N}$
  - si $p_i > 0$, il existe $2.n$ racines dans $]a_i, a_{i+1}[$, $n \in \mathbb{N}$
  - si $p_i = 0$, alors a_i ou a_{i+1} est racine (voire les deux).

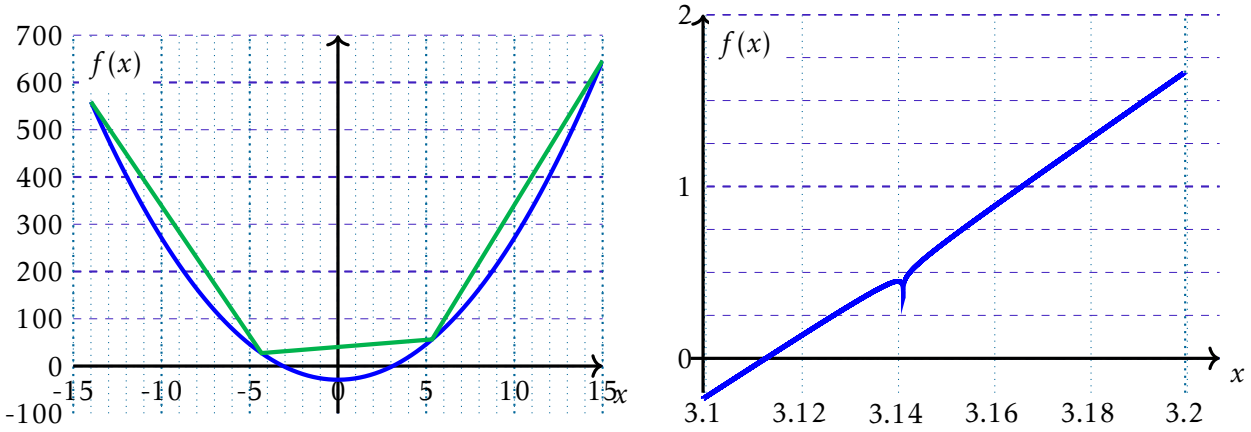
Évidemment, cet algorithme ne donne pas la valeur de n. Nous verrons plus loin l'utilité de ce test, notamment dans la recherche de la racine par dichotomie.

2.2.2 Représentations graphiques

Les représentations graphiques sont utiles parfois pour avoir une idée des lieux de convergence. Cela permet d'obtenir à vu d'œil un premier intervalle ou un premier candidat pour démarrer une méthode numérique.

Cependant, il convient aussi de se méfier des représentations graphiques. Pour cela, il est préférable d'observer attentivement l'expression de la fonction et son domaine de définition.

Exemple : $f(x) = 3.x^3 + \dfrac{1}{\pi^4}.\ln\left[(\pi - x)^2\right] - 29$



Courbes tracées entre -14 et 15. La courbe verte n'ayant que 4 points, on ne constate pas d'intersection avec l'axe des abscisses.

Avec le zoom sur l'intervalle $[3,1;3,2]$, il semble encore qu'il n'y ait qu'une racine sur l'intervalle. Or $\lim\limits_{x \mapsto \pi} f(x) = -\infty$.

Par continuité sur les intervalle $[3,14;\pi[$ et $[\pi;3,15]$, la fonction f admet donc 3 racines dans l'intervalle $[3,1;3,2]$.

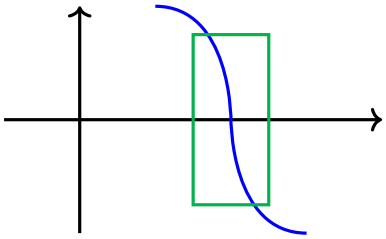
2.2.3 Critères de convergence

Les méthodes numériques ne permettent pas d'obtenir une réponse formelle à un problème. En revanche, elles permettent d'approcher la solution d'un problème avec une précision fonction du type de stockage des nombres (cf premier semestre) et du calcul numérique nécessaire pour obtenir cette solution (cf la sensibilité de certains calculs aux erreurs d'arrondis).

Pour une erreur normée en abscisse (fonction de l'*ordre de grandeur* de x), une tolérance de 10^{-8} est raisonnable. En revanche, chercher à obtenir 10^{-16} est a priori non nécessaire et impossible.

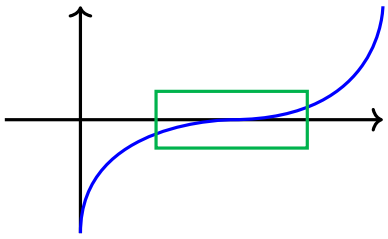
Dans le cas de la méthode par dichotomie, il est possible de faire un test de convergence sur la valeur de x et/ou sur la valeur de la fonction :

$$\frac{|b_n - a_n|}{|a_n| + |b_n|} < \varepsilon_x \quad \text{et/ou} \quad |f(c_n)| < \varepsilon_f$$



Fort gradient

Suivant le gradient (pente) de la fonction, un des deux tests sera privilégié. Pour un fort (faible) gradient, il convient de suivre la valeur de f (de x).



Faible gradient

En absence d'information, on peut suivre les deux (si la méthode donne une information sur l'erreur en x).

2.2.4 Ordre de convergence

Soit x_n la énième valeur calculée pour résoudre l'équation $f(x) = 0$ sur l'intervalle donné. Soit r la solution exacte de cette équation. On peut alors définir ε_n comme l'erreur en abscisse à l'itération n.

La méthode numérique est qualifiée de méthode à l'ordre p si : $\exists K \in \mathbb{R}^+ / \lim_{n\to\infty} \frac{\varepsilon_{n+1}}{\varepsilon_n^p} = K$.

- Si $p = 1$ et $K = 1$ la méthode est dite sous-linaire.
- Si $p = 1$ et $K \in]0, 1[$ la méthode est dite linéaire.
- Si $p > 1$, la méthode est qualifiée de super-linéaire.
- Si $p = 2$ la méthode est quadratique.

2.3 Rappel sur la méthode par dichotomie

2.3.1 Principe

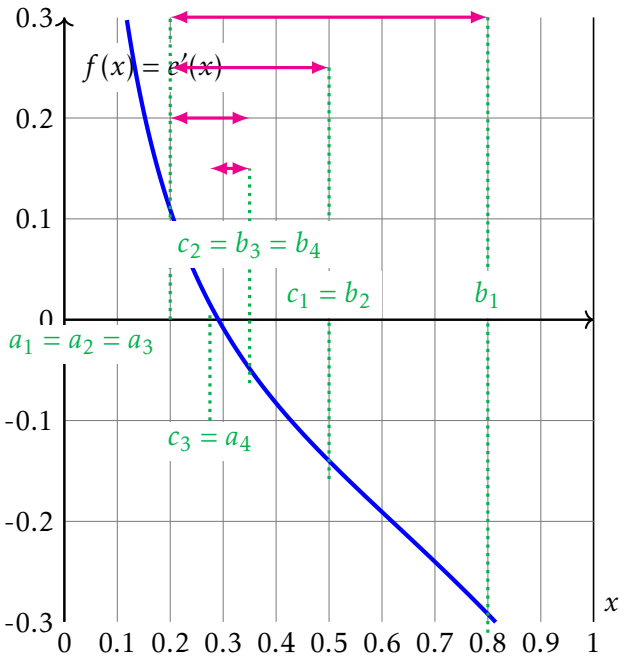
Le principe est de diviser l'intervalle en deux parts égales, de conserver la partie contenant la racine et d'y reproduire l'opération jusqu'à ce que le critère de convergence soit satisfait.

A partir d'un intervalle donné $[a, b]$, encadrant une racine de la fonction f étudiée :

- calculer le point c milieu de l'intervalle : $c = \frac{a+b}{2}$
- évaluer $p = f(a).f(c)$ puis test :
  - si $p > 0$, il n'y a pas de racine dans l'intervalle $[a, c]$. La racine est donc dans l'intervalle $[c, b]$. On donne alors à a la valeur de c
  - si $p < 0$, la racine est dans $[a, c]$. On donne alors à b la valeur de c
  - si $p = 0$, alors la racine est c. Ô Miracle,...
- test d'arrêt : Evaluer le critère de convergence
- recommencer si le critère de convergence n'est pas satisfait

2.3.2 Algorithme

Méthode par dichotomie pour la fonction f sur $[a,b]$ avec une tolérance ε :



Algorithm 1 Dichotomie

Dichotomie(f,a,b,ε)
entrée: f, une fonction, a et b un encadrement de la racine cherchée et ε une tolérance
résultat: $(bi-ai)/2$ tel que $bi-ai \leq 2.\varepsilon$
$ai, bi \leftarrow a, b$
$fa \leftarrow f(ai)$
tant que $bi-ai > 2.\varepsilon$ **faire**
  $ci \leftarrow (ai+bi)/2$
  $fc \leftarrow f(ci)$
  si $fa.fc \leq 0$ **alors**
    $bi \leftarrow ci$
  sinon
    $ai \leftarrow ci$
    $fa \leftarrow fc$
  fin si
fin tant que
renvoi: $\dfrac{ai+bi}{2}$

2.3.3 Convergence de la méthode

A l'itération n, l'erreur ϵ_n entre la solution exacte et la solution numérique calculée peut être majorée par $|\epsilon_n| < \dfrac{|b_n-a_n|}{2}$. Ainsi, la racine r de la fonction peut être encadrée : $c_n - \dfrac{b_n-a_n}{2} \leq r \leq c_n + \dfrac{b_n-a_n}{2}$

Soit $[a_n,b_n]$ l'intervalle à l'itération n. Définissons ε_n par $\varepsilon_n = b_n - a_n$.
Comme l'intervalle est divisé par deux à chaque itération, $|\epsilon_{n+1}| \leq \varepsilon_{n+1} = \dfrac{\varepsilon_n}{2} = \dfrac{\varepsilon_1}{2^n}$. La convergence est donc linéaire.

Pour test d'arrêt, il est alors possible de prendre :

- soit un encadrement de la solution exacte : $b_n - a_n \leq \varepsilon$
- soit la valeur de la fonction au point calculé : $|f(c_n)| < \varepsilon$

Le nombre d'itération n pour obtenir un erreur inférieur à ε est tel que :

$$\varepsilon \leq \frac{b-a}{2^n} \quad \Rightarrow \quad 2^n \leq \frac{b-a}{\varepsilon} \quad \Rightarrow \quad n.\ln(2) \leq \ln\left(\frac{b-a}{\varepsilon}\right) \quad \Rightarrow \quad n \geq \frac{1}{\ln 2}.\ln\left(\frac{b-a}{\varepsilon}\right)$$

Avec $\varepsilon = 10^{-p}$ (resp. $\varepsilon = 2^{-p}$) pour une approche décimal (resp. binaire) :

$$n \geq \frac{\ln(b-a)+p.\ln(10)}{\ln 2} \qquad \left(\text{resp. } n \geq p + \frac{\ln(b-a)}{\ln 2}\right) \text{ la compléxité est donc logarithmique}$$

La vitesse de convergence est lente mais la méthode est robuste. Si :

- la fonction f est définie et continue sur l'intervalle $[a,b]$
- la fonction n'admet qu'une seule racine sur l'intervalle $[a,b]$

alors la méthode converge.

2.4 Méthode de Newton

2.4.1 Principe

Les méthodes de Schröder sont basés sur les n premières dérivées d'une fonction f pour une méthode d'ordre $n+1$. Dans le cas de la méthode de Newton, on n'utilise que la dérivée première.

Rappel Si la fonction est de classe C^1 sur l'intervalle $[a,b]$ alors, le développement de Taylor à l'ordre 1 donne :

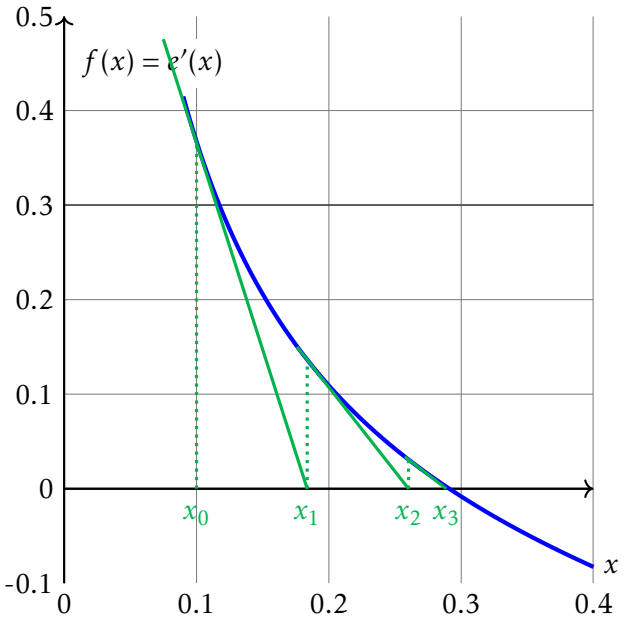
$$f(b) = f(a) + f'(a).(b-a) + o(b-a)$$

A partir d'un point x_0, la méthode consiste à rechercher le point suivant x_1 en le supposant racine de la fonction et en négligeant le terme $o(b-a)$ dans le développement de Taylor à l'ordre 1. Ainsi :

$$0 = f(x_1) = f(x_0) + f'(x_0).(x_1 - x_0) \quad \Rightarrow \quad x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$$

Soit f de classe C^1 sur $I = [a,b]$:

- choisir un premier candidat x_0
- test d'arrêt : Tester le critère de convergence $|f(x_0)| < \epsilon$ et $x_0 \notin I$
- si le critère n'est pas atteint, calculer le nouveau candidat $x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$
- reprendre depuis le test d'arrêt



2.4.2 Algorithme

Méthode de Newton pour la fonction f sur $[a,b]$ avec une tolérance ε :

Algorithm 2 Méthode de Newton

```
xi ← x0
fx ← f(xi)
tant que |fx| > ε faire
    fpx ← f'(xi)
    si fpx=0 alors
        break
    fin si
    xi ← xi − fx/fpx
    fx ← f(xi)
fin tant que
renvoi: xi
```

2.4.3 Convergence de la méthode

Soit r la racine de la fonction f sur l'intervalle d'étude et x_n, la énième valeur calculée par itération de Newton. Notons alors $\varepsilon_n = x_n - r$. Le développement de Taylor à l'ordre 2 au voisinage de r donne :

$$\begin{aligned} f(x_n) &= \cancel{f(r)} + f'(r).\varepsilon_n + f''(r).\frac{\varepsilon_n^2}{2} + o(\varepsilon_n^2) \\ f'(x_n) &= f'(r) + f''(r).\varepsilon_n + o(\varepsilon_n) \end{aligned}$$

$$\text{alors} \quad \varepsilon_{n+1} = x_{n+1} - r = x_n - r - \frac{f(x_n)}{f'(x_n)} = \varepsilon_n - \frac{f'(r).\varepsilon_n + f''(r).\frac{\varepsilon_n^2}{2} + o(\varepsilon_n^2)}{f'(r) + f''(r).\varepsilon_n + o(\varepsilon_n)} = \frac{\varepsilon_n^2}{2}.\frac{f''(r)}{f'(r)} + o(\varepsilon_n^2)$$

La convergence est donc quadratique.

2.4.4 Limites et précautions

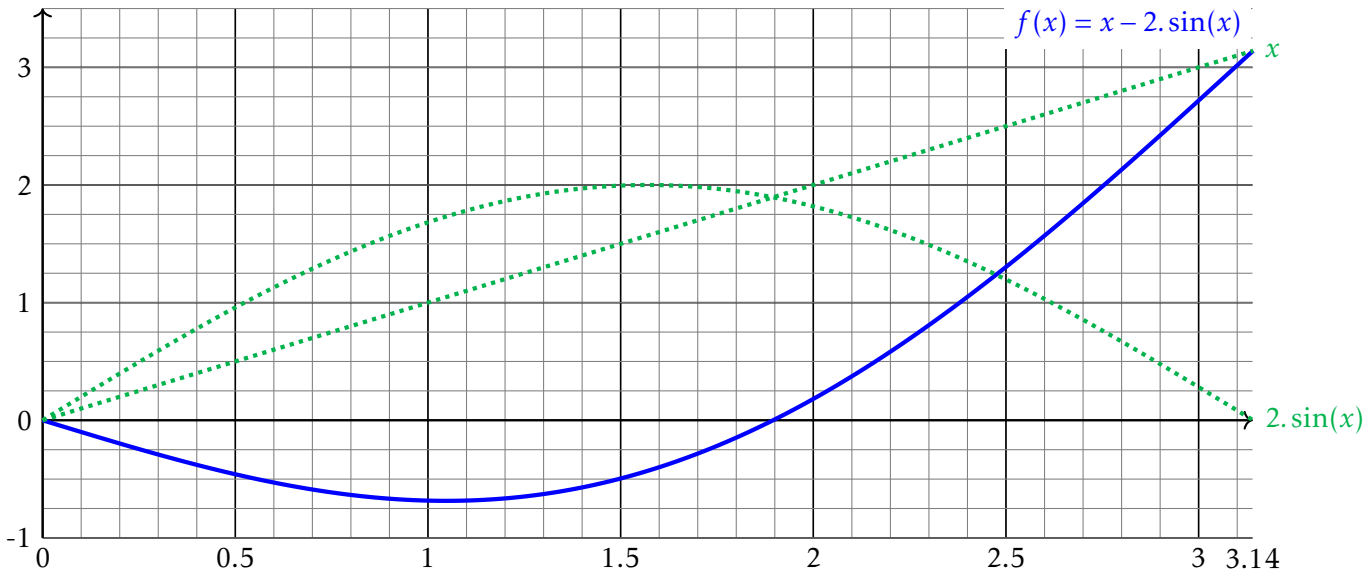
La méthode est bien adapté si la valeur de x_0 est proche de la racine r de la fonction f et si la dérivée n'est pas trop faible (pente trop horizontale).
Soit la fonction $f(x) = x - 2.\sin(x)$. La dérivée $f'(x) = 1 - 2.\cos(x)$ s'annule pour $x = \frac{\pi}{3} \approx 1,0472$. Prendre comme valeur $x_0 = \frac{\pi}{3}$ conduit tout de suite au crash. La dérivée étant nulle, x_1 ne pourra pas être calculé. Prendre $0 < x_0 < \frac{\pi}{3}$ conduit à la racine $r_0 = 0$. Pour trouver la deuxième racine $r_1 \approx 1,9$, il convient de prendre une valeur de départ $x_0 > \frac{\pi}{3}$.

n	x_n	$f(x_n)$
0	1,1	−0,6824147
1	8,452992	6,801205
2	5,256414	6,967679
3	203,384184	201,922795
⋮	⋮	⋮
57	−0,380713	0,362452
58	0,042317	−0,042292
59	−0,000051	0,000051
60	0,000000	−0,000000

Cependant, prendre pour valeur $x_0 = 1,1$ ou $x_0 = 1,2$ conduit dans le premier cas à r_0 et dans le deuxième à r_1, même si la fonction est monotone sur $\left[\frac{\pi}{3}, \pi\right]$ et que $\frac{\pi}{3} < 1,1 < 1,2 < \pi$.

Notons dans les deux cas, la convergence quadratique à proximité de r.

n	x_n	$f(x_n)$
0	1,2	−0,6640782
1	3,612334	4,519429
2	1,988080	0,159694
3	1,899879	0,007200
4	1,895505	0,000018
5	1,895494	0,000000



Il est donc préférable que la fonction soit monotone sur l'intervalle d'étude et que la première valeur x_0 soit proche de r.

Lorsque l'algorithme patine ou que l'itération conduit à une valeur hors du domaine de définition, il est possible d'utiliser une méthode par dichotomie et reprendre le schéma de Newton près de la racine.

2.4.5 Fausse position

2.4.5.1 Principe

La méthode de Newton étant basée sur un développement de Taylor à l'ordre 1, il est nécessaire de calculer la dérivée première.

La méthode de la fausse position est basée sur une estimation de la dérivée à partir des deux points précédents. Ce n'est pas la valeur exacte comme dans le schéma de Newton. Nous avons alors :

$$f'(x_n) \approx \frac{f(x_n)-f(x_{n-1})}{x_n-x_{n-1}} \quad \Rightarrow \quad x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \approx x_n - f(x_n).\frac{x_n-x_{n-1}}{f(x_n)-f(x_{n-1})} = \frac{x_{n-1}.f(x_n)-x_n.f(x_{n-1})}{f(x_n)-f(x_{n-1})}$$

Soit f définie sur $\mathrm{I}=[a,b]$:

- choisir un premier couple candidat $(x_0,x_1)\in \mathrm{I}^2$
- test d'arrêt : tester le critère de convergence $|f(x_1)|<\varepsilon$ ou $x_1\notin \mathrm{I}$
- si le critère n'est pas atteint :
  - calculer $f(x_1)$
  - si $f(x_1)=f(x_0)$ soit arrêt, soit choisir une autre valeur de x_0
  - calculer le nouveau candidat $x_2 = x_1 - f(x_1).\dfrac{x_1-x_0}{f(x_1)-f(x_0)}$
- reprendre depuis le test d'arrêt avec : si $f(x_n)=f(x_{n-1})$ soit arrêt, soit prendre x_{n_2} à la place de x_{n_1}

2.4.5.2 Convergence de la méthode

Soit r la racine de la fonction f sur l'intervalle d'étude et x_n, la énième valeur calculée par la méthode de la fausse position. Notons alors $\varepsilon_n = x_n - r$. Le développement de Taylor à l'ordre 2 au voisinage de r donne :

$$\begin{aligned} f(x_n) &= \cancel{f(r)} + f'(r).\varepsilon_n + f''(r).\frac{\varepsilon_n^2}{2} + o(\varepsilon_n^2) \\ f(x_{n-1}) &= \cancel{f(r)} + f'(r).\varepsilon_{n-1} + f''(r).\frac{\varepsilon_{n-1}^2}{2} + o(\varepsilon_{n-1}^2) \end{aligned}$$

$$\begin{aligned} \text{alors} \quad \varepsilon_{n+1} = x_{n+1} - r &= \frac{x_{n-1}.f(x_n)-x_n.f(x_{n-1})}{f(x_n)-f(x_{n-1})} - r = \frac{\varepsilon_{n-1}.f(x_n)-\varepsilon_n.f(x_{n-1})}{f(x_n)-f(x_{n-1})} \\ &= \frac{f''(r).\left(\frac{\varepsilon_n^2}{2}.\varepsilon_{n-1} - \frac{\varepsilon_{n-1}^2}{2}.\varepsilon_n\right) + o(\varepsilon_n^2) - o(\varepsilon_{n-1}^2)}{f'(r).(\varepsilon_n-\varepsilon_{n-1}) + o(\varepsilon_n) - o(\varepsilon_{n-1})} \approx \frac{f''(r).\varepsilon_n.\varepsilon_{n-1}.\left(\frac{\varepsilon_n}{2} - \frac{\varepsilon_{n-1}}{2}\right)}{f'(r).(\varepsilon_n-\varepsilon_{n-1})} = \alpha.\varepsilon_n.\varepsilon_{n-1} \end{aligned}$$

Nous cherchons à connaître l'ordre p de convergence, i.e, p tel que $\varepsilon_{n+1} = \varepsilon_n^p$. Or :

$$\varepsilon_{n+1} \equiv \varepsilon_n.\varepsilon_{n-1} \quad \Rightarrow \quad \varepsilon_n^p \equiv \varepsilon_n.\varepsilon_n^{\frac{1}{p}} \quad \Rightarrow \quad p = 1+\frac{1}{p} \quad \Rightarrow \quad p^2-p-1=0 \quad \Rightarrow \quad p = \frac{1+\sqrt{5}}{2} \approx 1,62 < 2$$

La convergence est donc super-linéaire ($p>1$) mais inférieure à la convergence quadratique ($p=2$) de la méthode de Newton. En revanche, elle ne nécessite pas le calcul de la dérivée première qui peut parfois être très lourd.

2.4.6 Avec le module scipy.optimize

Pour résoudre un problème du type $f(x) = 0$, le plus simple est d'utiliser le module `scipy.optimize` avec la méthode `newton`. Ainsi, `sciop.newton(f, x0)` permet de déterminer un zéro de la fonction `f` en partant de `x0`. Il n'est pas nécessaire de donner la dérivée `fp` de `f` mais il est possible de le faire `sciop.newton(f, x0, fp)`.

```
def f(x):
    return x**2/2 - 1

def fp(x):
    return x

sol = sciop.newton(f, 3, fp)
```

Rappel il n'est pas nécessaire de déclarer les fonctions avant d'utiliser `newton`. La fonction `lambda` permet de faire lors de l'appel.

```
>>> sciop.newton(lambda x: x**2/2 - 1, 3)
1.4142135623730971
```

2.4.7 Méthode de Newton-Raphson

La méthode de Newton-Raphson est une généralisation de la méthode de Newton pour les fonctions de plusieurs variables. Il s'agit de résoudre un système de n équations à n inconnues :

$$\vec{F}(\vec{X}) = \vec{0} \quad \Leftrightarrow \quad \begin{bmatrix} F_1(x_1,\dots,x_i,\dots,x_n) \\ \vdots \\ F_i(x_1,\dots,x_i,\dots,x_n) \\ \vdots \\ F_n(x_1,\dots,x_i,\dots,x_n) \end{bmatrix} = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

La formule de Taylor à l'ordre 1 donne pour chaque F_i :

$$F_i(\vec{x}+\vec{\delta x}) = F_i(\vec{x}) + \sum_{j=1}^{n} \frac{\partial F_i(\vec{x})}{\partial x_j}.\delta x_j + o(\vec{\delta x}),\ \forall i \in [\![1,n]\!]$$

En introduisant alors la matrice jacobienne $\mathbb{J}$ définie par

$$[\mathbb{J}]_{(i,j)} = J_{i,j} = \frac{\partial F_i(\vec{x})}{\partial x_j} \quad \Leftrightarrow \quad \mathbb{J} = \begin{bmatrix} \frac{\partial F_1(\vec{x})}{\partial x_1} & \cdots & \frac{\partial F_1(\vec{x})}{\partial x_j} & \cdots & \frac{\partial F_1(\vec{x})}{\partial x_n} \\ \vdots & \ddots & \vdots & & \vdots \\ \frac{\partial F_i(\vec{x})}{\partial x_1} & \cdots & \frac{\partial F_i(\vec{x})}{\partial x_j} & \cdots & \frac{\partial F_i(\vec{x})}{\partial x_n} \\ \vdots & & \vdots & \ddots & \vdots \\ \frac{\partial F_n(\vec{x})}{\partial x_1} & \cdots & \frac{\partial F_n(\vec{x})}{\partial x_j} & \cdots & \frac{\partial F_n(\vec{x})}{\partial x_n} \end{bmatrix}$$

la formule générale du développement limité à l'ordre 1 de $\vec{F}$ au voisinage de $\vec{x}$ devient :

$$\vec{F}(\vec{x}+\vec{\delta x}) = \vec{F}(\vec{x}) + \mathbb{J}(\vec{x}).\vec{\delta x} + o(\vec{\delta x})$$

L'algorithme de Newton - Raphson s'obtient en cherchant $\overrightarrow{X_{n+1}}$ solution du problème $\vec{F}(\vec{x}) = \vec{0}$. Le développement de Taylor, conduit à :

$$\cancel{\vec{F}(\overrightarrow{X_{n+1}})} = \vec{F}(\overrightarrow{X_n}) + \mathbb{J}(\overrightarrow{X_n}).\left(\overrightarrow{X_{n+1}} - \overrightarrow{X_n}\right) + o\left(\overrightarrow{X_{n+1}} - \overrightarrow{X_n}\right) \quad \text{d'où} \quad \overrightarrow{X_{n+1}} = \overrightarrow{X_n} - \mathbb{J}^{-1}(\overrightarrow{X_n}).\vec{F}(\overrightarrow{X_n})$$

Chaque itération demande alors le calcul de l'inverse de la matrice Jacobienne…

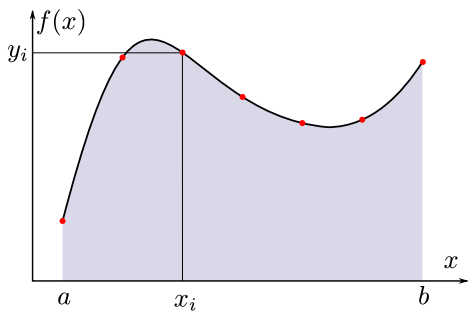
2.5 Dilemme robustesse/rapidité

Nous avons vu différentes méthodes de résolution de l'équation $f(x) = 0$. Que choisir ?

- La méthode la plus naïve est la dichotomie. A partir d'une fonction continue sur un intervalle $[a, b]$, avec $f(a).f(b) \leq 0$, la recherche de x tel que $f(x) = 0$ est simple et robuste mais lente (convergence linéaire). Si l'on souhaite un programme simple et robuste, la dichotomie est une solution.
- Si on cherche une valeur avec précision, une convergence quadratique sera préférée (méthode de Newton). A ce moment là, il convient d'avoir une idée approximative de la solution pour proposer un premier candidat au voisinage de cette solution de sorte que la fonction y présente de bonnes propriétés. Si on ne souhaite pas calculer la dérivée de la fonction, on peut alors utiliser la méthode de la fausse position.
- Dans le cas où l'on cherche rapidité et stabilité, on peut utiliser la méthode par dichotomie dans une premier temps pour localiser le zéro de la fonction, puis appliquer un algorithme de Newton. En cas d'instabilité de l'algorithme de Newton, il est toujours possible de réutiliser une méthode par dichotomie.
- Enfin, mentionnons qu'il existe aussi d'autres types d'algorithme, comme les algorithmes génétiques pour trouver *les* différentes solutions du problème.

Chapitre 3

Interpolation, intégration et dérivation numérique



Sommaire

3.1 Interpolation et approximation de fonctions

Définition : Interpolation

Méthode qui consiste à déterminer une fonction (dans un ensemble donné), passant par un certain nombre de points imposés.

L'interpolation est utile lorsqu'une loi est donnée à partir d'une liste de points et qu'il est nécessaire d'évaluer le résultat en des points intermédiaires. Nous verrons par la suite que les fonctions d'interpolations sont aussi à la base de l'intégration et de la dérivation numérique.

Définition : Approximation

Méthode qui consiste à déterminer une fonction passant "au mieux" à proximité des points donnés.

Une approximation est utile lorsque une loi théorique est recherchée à partir de points de mesure (nombreux, mais entachés de bruit de mesure).

3.1.1 Interpolation polynomiale

Définition : Interpolation polynomiale

Interpolation où la fonction est recherchée dans l'ensemble des polynômes.

En exprimant le polynôme dans une base, par exemple dans la base canonique $(1, x, x^2, \ldots, x^{n-1})$ (mais ce n'est pas le seul choix possible) :

$$\mathrm{P}(x) = a_0 + a_1.x + a_2.x^2 + \cdots + a_{n-1}.x^{n-1}$$

L'opération consiste à déterminer les n composantes a_i du polynôme passant par les points imposés. Chaque point de passage constituant une condition scalaire sur les coefficients, il existe une solution unique si la dimension n de la base correspond au nombre de points.

Les composantes a_i sont solutions du système de n équations, pour n points imposés (x_i, y_i) :

$$y_i = a_0 + a_1.x_i + a_2.x_i^2 + \cdots + a_{n-1}.x_i^{n-1} \quad \text{avec} \quad i \in [\![1..n]\!]$$

La base polynomiale de Lagrange est plus pratique pour l'expression directe du polynôme interpolant à partir des points, mais s'avère plus lourde à évaluer pour l'ordinateur. Le polynôme interpolant n points, s'écrit directement dans la base de Lagrange sous la forme :

$$\mathrm{P}(x) = \sum_{i=1}^{n} y_i . \prod_{j=1 \& j \neq i}^{n} \frac{x - x_j}{x_i - x_j}$$

Exemple : Pour 3 points $\{(x_1, y_1), (x_2, y_2), (x_3, y_3)\}$, le polynôme interpolant s'écrit :

$$\mathrm{P}(x) = y_1 . \frac{(x-x_2).(x-x_3)}{(x_1-x_2).(x_1-x_3)} + y_2 . \frac{(x-x_1).(x-x_3)}{(x_2-x_1).(x_2-x_3)} + y_3 . \frac{(x-x_1).(x-x_2)}{(x_3-x_1).(x_3-x_2)}$$

L'expression se simplifie pour chaque (x_i, y_i), ce qui permet de vérifier que le polynôme P passe bien par les points.

L'interpolation polynomiale n'est cependant pas idéale dès que le nombre de point augmente : le polynôme interpolant peut alors présenter des oscillations entre les points (phénomène de Runge, Fig 3.12). L'interpolation peut alors être localement très éloignée des points. Il est souvent plus adapté d'interpoler par morceaux.

3.1.2 Interpolation par morceaux

Pour éviter les oscillations sur certaines fonctions, il est plus satisfaisant de réaliser une interpolation polynomiale de faible degré mais par morceaux.

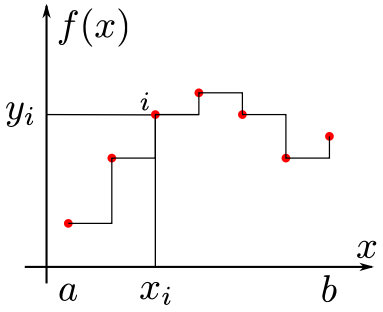


Figure 3.1 – Interpolation de degré 0.

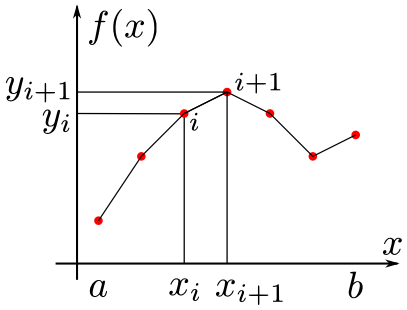


Figure 3.2 – Interpolation de degré 1.

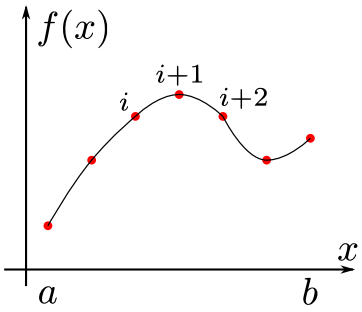


Figure 3.3 – Interpolation de degré 2.

3.1.2.1 Interpolation par morceaux de degrés 0

Il s'agit de considérer qu'entre deux points, la valeur de la fonction vaut une constante, égale à la valeur du point précédent, du point suivant ou encore égale à la moyenne des valeurs des points encadrant.

Cette interpolation est très rudimentaire mais elle peut être suffisante si le nombre de point est très important.

La fonction interpolante n'est pas continue bien entendu.

3.1.2.2 Interpolation linéaire par morceaux (degré 1)

Une loi affine $(a.x+b)$ est adoptée entre deux points successifs, passant évidement par les deux points :

$$\forall x \in [x_i, x_{i+1}], \qquad y = y_i + (x - x_i).\frac{y_{i+1} - y_i}{x_{i+1} - x_i}$$

La fonction obtenue est continue mais sa dérivée ne l'est pas.

3.1.2.3 Interpolation quadratique par morceaux (degré 2)

Une loi parabolique $(a.x^2 + b.x + c)$ est adoptée sur chaque intervalle regroupant 3 points successifs, passant par les 3 points.

La fonction obtenue est continue mais sa dérivée ne l'est pas car elle présente des discontinuités de la pente entre chaque portion de parabole. L'interpolation spline résout ce problème et assure une continuité C_1.

3.1.2.4 Autres méthodes par morceaux

3.1.2.5 Choix du degré du polynôme d'interpolation

Pour limiter les oscillations (phénomène de Runge), nous avons déjà indiqué qu'il faut éviter les polynômes de degrés trop élevés (Fig 3.12). Par ailleurs, la qualité de la courbe devient très pauvre pour des degrés trop faibles, ce qui pousse à utiliser préférentiellement les degrés 1, 2 ou 3.

Il existe bien d'autres méthodes pour interpoler un ensemble de points, pour une fonction ou plus généralement pour une courbe du plan, ou encore pour des surfaces, des champs scalaires ou vectoriels (plans ou volumiques).

Pour les courbes dans le plan, citons l'interpolation d'Hermite cubique, qui est une interpolation de degrés 3 par morceau assurant la continuité de la dérivée aux extrémités des morceaux, ou encore les splines cubiques, qui est une interpolation cubique (degrés 3) par morceaux avec des conditions de continuité des deux premières dérivées aux extrémités des morceaux, ce qui conduit à une fonction de classe C_2.

De même, les courbes de Bezier permettent d'interpoler des courbes ou des surfaces par des expressions polynomiales.

On retrouve alors un équivalent des cerces (Fig 3.4) utilisées en menuiserie, carrosserie...

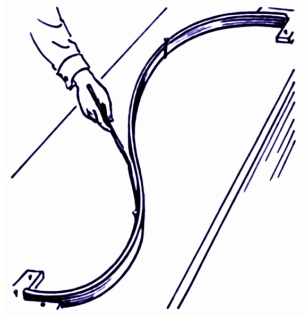


Figure 3.4 – Interpolation à l'aide d'une cerce en menuiserie

Néanmoins, soulignons que ce choix dépend aussi de la régularité de la courbe à interpoler car le choix d'un degré 2 ou 3 suppose une continuité de la dérivée sur la courbe d'origine. Si la courbe présente des discontinuités ou des ruptures de pentes, le degré 1 est à privilégier.

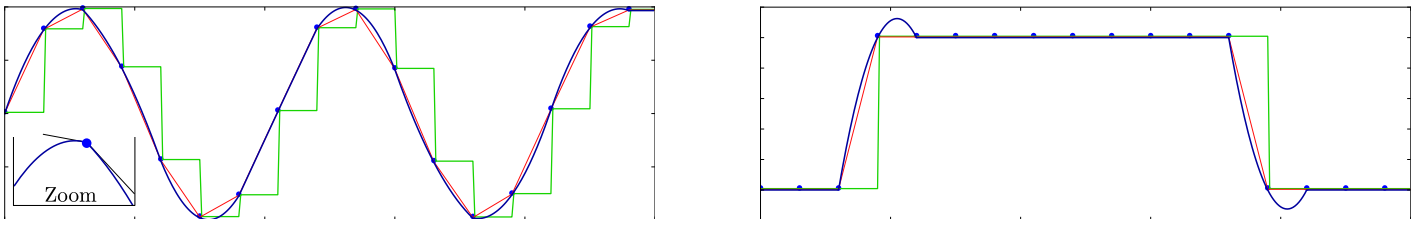


Figure 3.5 – Interpolations d'ordres 0, 1 et 2 pour une sinusoïde (fonction régulière) et un signal carré (fonction discontinue).

La Fig 3.5 montre l'interpolation par morceaux d'un signal sinusoïdal (C_∞) et d'un signal carré (discontinu). Le choix d'un degré 2 est bien approprié pour la première courbe mais pas pour la seconde !

3.1.3 Approximation polynomiale par moindres carrés

Dans le cas de l'approximation polynomiale, on cherche à minimiser la distance (en norme 2) entre un polynôme de degré m et les n points imposés $(x_i, y_i)_1^n$.

$$\text{Si } P(x) = a_0 + a_1.x + \cdots + a_m.x^m \quad \text{alors} \quad E_{rr}(a_0, \ldots, a_m) = \frac{1}{2}\sum_{i=1}^{n}(P(x_i) - y_i)^2$$

Minimiser E_{rr} suivant les paramètres $(a_j)_{j=0}^m$, revient à résoudre le système suivant :

$$\begin{bmatrix} \frac{\partial E_{rr}}{\partial a_0} \\ \vdots \\ \frac{\partial E_{rr}}{\partial a_m} \end{bmatrix} = \begin{bmatrix} 0 \\ \vdots \\ 0 \end{bmatrix} \quad \Rightarrow \quad \forall j \in [\![0, m]\!] \ \sum_{i=1}^{n}\Big(P(x_i) - y_i\Big).\frac{\partial P(x_i)}{\partial a_j} = \sum_{i=1}^{n}\Big(P(x_i) - y_i\Big).x_i^j = 0$$

La minimisation de D revient alors à résoudre le système suivant :

$$\underbrace{\sum_{i=1}^{n}\begin{bmatrix} x_i^0.x_i^0 & \dots & x_i^m.x_i^0 \\ \vdots & \ddots & \vdots \\ x_i^0.x_i^m & \dots & x_i^m.x_i^m \end{bmatrix}}_{\mathrm{M}}.\underbrace{\begin{bmatrix} a_0 \\ \vdots \\ a_m \end{bmatrix}}_{a} = \underbrace{\sum_{i=1}^{n} y_i.\begin{bmatrix} x_i^0 \\ \vdots \\ x_i^m \end{bmatrix}}_{b} \quad \Rightarrow \quad a = \mathrm{M}^{-1}.b \quad \text{si M est inversible.}$$

3.2 Intégration numérique

3.2.1 Principe, degré et ordre de la méthode

L'intégration numérique (ou quadrature) consiste à intégrer (de façon approchée) une fonction sur un intervalle borné $[a,b]$, c'est-à dire calculer l'aire sous la courbe représentant la fonction, à partir d'un calcul ou d'une mesure en un nombre fini n de points.

La répartition des points en abscisse est généralement uniforme (échantillonnage à pas constant $h = \dfrac{b-a}{n}$) mais il existe des méthodes à pas variable, ou encore à pas adaptatif.

L'intégration des polynômes étant très simple, l'opération consiste généralement à construire une interpolation polynomiale (de degré plus ou moins élevé) par morceaux (intégration composée) puis d'intégrer le polynôme sur chaque morceau.

Les méthodes de quadrature élémentaires composées sont de la forme :

$$\int_a^b f(x).dx = \sum_{i=0}^{n-1}\int_{x_i}^{x_{i+1}} f(x).dx \approx \frac{b-a}{n}.\sum_{i=0}^{n-1}\sum_{j=0}^{m}\omega_j.f(\lambda(i,j)) = \mathrm{I}(f,n,m)$$

$$\text{avec} \quad x_i = a + i.\frac{b-a}{n} \quad ; \quad \lambda(i,j) \in [x_i, x_{i+1}] \quad \text{et} \quad \sum_{j=0}^{m}\omega_j = 1$$

Définition : Méthode de degré N

> *Méthode pour laquelle la formule approchée est exacte pour tout polynôme de degré au plus* N *et inexacte pour au moins un polynôme de degré* N + 1.

Si on appelle $\mathrm{E}_{rr}(f,n,\mathrm{N})$ la différence entre la solution exacte de l'intégrale et sa valeur approchée par une méthode d'ordre N sur n segments

$$\mathrm{E}_{rr}(f,n,\mathrm{N}) = \left|\int_a^b f(x).dx - \mathrm{I}(f,n,m)\right| = \left|\int_a^b f(x).dx - \frac{b-a}{n}.\sum_{i=0}^{n-1}\sum_{j=0}^{m}\omega_j.f(\lambda(i,j))\right|$$

alors on peut montrer, moyennant une régularité suffisante de f, qu'il existe $\mathrm{K} \in \mathbb{R}$ tel que : $\mathrm{E}_{rr}(f,n,\mathrm{N}) \leq \dfrac{\mathrm{K}}{n^{\mathrm{N}+1}}$.
Une méthode de degré N est donc d'ordre au moins N + 1.

La précision de l'intégration numérique peut ainsi s'améliorer en augmentant le nombre de points n (en diminuant le pas d'échantillonnage h) ou en augmentant le degré de l'interpolation polynomiale (sous réserve de bonnes propriétés de continuité de la courbe).

3.2.2 Méthodes d'intégration composée

3.2.2.1 Méthode des rectangles (degré 0)

L'intégration la plus simple est celle de degré 0 où le polynôme interpolateur sur chaque segment est une constante ($m = 0$ et $\omega_0 = 1$) prise soit à gauche, soit à droite de l'intervalle d'intégration. L'intégrale est donc approchée par des rectangles pour calculer l'aire sous la courbe :

$$\int_a^b f(x).dx \approx \frac{b-a}{n}.\sum_{i=0}^{n-1} f(\lambda(i,0)) \quad \Rightarrow \quad E_{rr}(f,n,0) \leq \frac{K}{n} \quad \text{avec} \quad K = \frac{\sup|f'|}{2}.|b-a|^2$$

Rectangles à gauche (degré 0)

Dans le cas de la formule des rectangles à gauche, on pose $\lambda(i,0) = x_i$:

$$\int_a^b f(x).dx \approx \frac{b-a}{n}.\sum_{i=0}^{n-1} f(x_i) = \frac{b-a}{n}.\sum_{i=0}^{n-1} y_i$$

Rectangles à droite (degré 0)

Dans le cas de la formule des rectangles à droite, on pose $\lambda(i,0) = x_{i+1}$:

$$\int_a^b f(x).dx \approx \frac{b-a}{n}.\sum_{i=0}^{n-1} f(x_{i+1}) = \frac{b-a}{n}.\sum_{i=0}^{n-1} y_{i+1}$$

3.2.2.2 Méthode point milieu (degré 1)

La méthode du point milieu consiste à considérer la fonction interpolante constante sur chaque intervalle $[x_i, x_{i+1}]$ et égale à la valeur prise par le point au milieu de l'intervalle $\lambda(i,0) = \frac{x_i + x_{i+1}}{2}$ (Fig 3.6). La valeur approchée de l'intégrale s'écrit alors :

$$I(f,n,0) = \frac{b-a}{n}.\sum_{i=0}^{n-1} f\left(\frac{x_i + x_{i+1}}{2}\right) \quad \Rightarrow \quad E_{rr}(f,n,0) \leq \frac{K}{n^2} \quad \text{avec} \quad K = \frac{\sup|f''|}{24}.|b-a|^3$$

La méthode est d'ordre 3. En effet, la méthode du point milieu est exacte pour les polynômes de degré 1.

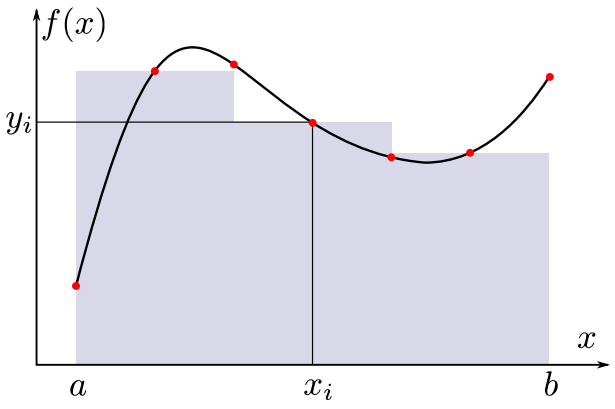


Figure 3.6 – Intégration au point milieu.

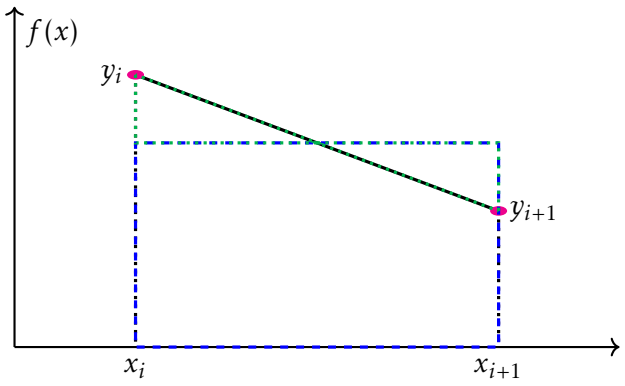


Figure 3.7 – Méthode point milieu - degré 1

3.2.2.3 Méthode trapèze (degré 1)

La méthode du trapèze s'appuie sur une interpolation linéaire entre chaque point (Fig 3.8). La valeur approchée de l'intégrale s'écrit alors :

$$\begin{aligned} \mathrm{I}(f,n,1) &= \frac{b-a}{n}.\sum_{i=0}^{n-1}\sum_{j=0}^{1}\omega_j.f(\lambda(i,j)) \quad \text{avec} \quad \omega_0=\omega_1=\frac{1}{2} \quad \text{et} \quad \lambda(i,0)=x_i\,;\,\lambda(i,1)=x_{i+1} \\ &= \frac{b-a}{n}.\sum_{i=0}^{n-1}\frac{y_i+y_{i+1}}{2} = \frac{b-a}{n}.\left(\frac{y_0+y_n}{2}+\left(\sum_{i=2}^{n-1}y_i\right)\right) \quad \Rightarrow \quad \mathrm{E}_{rr}(f,n,1)\le\frac{\mathrm{K}}{n^2} \quad \text{avec} \quad \mathrm{K}=\frac{\sup|f''|}{12}.|b-a|^3 \end{aligned}$$

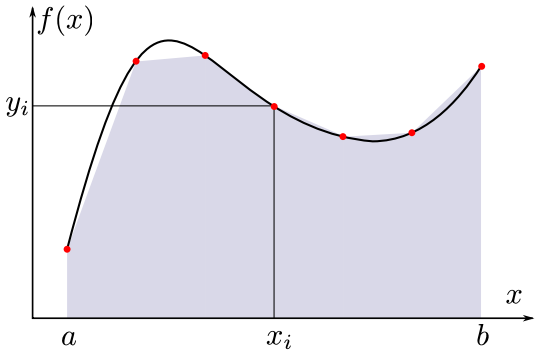


Figure 3.8 – Intégration par la méthode trapèze.

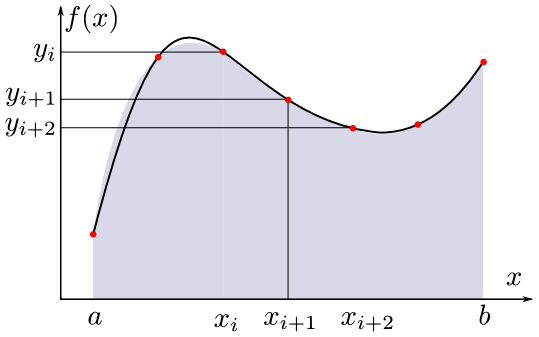


Figure 3.9 – Intégration par la méthode de Simpson.

Cette expression est à nouveau similaire au calcul approché réalisé au paragraphe précédent. La méthode du point milieu est au final plus précise que celle des trapèzes.

3.2.2.4 Méthode de Simpson (degré 2)

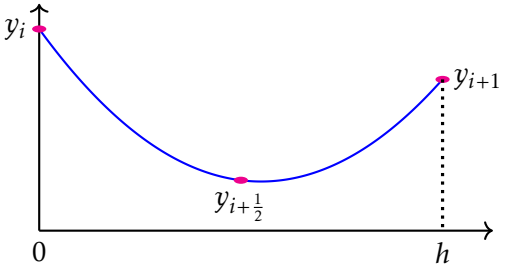
La méthode de Simpson s'appuie sur une interpolation quadratique $(a.x^2+b.x+c)$ sur chaque intervalle $[x_i,x_{i+1}]$: (Fig 3.9).

$$\int_a^b f(x).dx \approx \frac{b-a}{n}.\sum_{i=0}^{n-1}\sum_{j=0}^{2}\omega_j.f(\lambda(i,j)) = \mathrm{I}(f,n,2)$$

$$\text{avec} \quad \lambda(i,0)=x_i\,;\,\lambda(i,1)=x_{i+\frac{1}{2}}=\frac{x_i+x_{i+1}}{2} \quad \text{et} \quad \lambda(i,2)=x_{i+1}$$

En appliquant une translation de l'intervalle $[x_i,x_{i+1}]$ sur l'intervalle $[0,h]$, l'interpolation s'écrit :

$$\begin{cases} y_i = c \\ y_{i+\frac{1}{2}} = a.\dfrac{h^2}{4}+b.\dfrac{h}{2}+c \\ y_{i+1} = a.h^2+b.h+c \end{cases} \quad \Rightarrow \quad \begin{cases} a = \dfrac{2.y_i+2.y_{i+1}-4.y_{i+\frac{1}{2}}}{h^2} \\ b = \dfrac{4.y_{i+\frac{1}{2}}-y_{i+1}-3.y_i}{h} \\ c = y_i \end{cases}$$



L'intégrale sur $[0,h]$ vaut alors :

$$\begin{aligned} \int_0^h (a.x^2+b.x+c).dx &= \left[a.\frac{x^3}{3}+b.\frac{x^2}{2}+c\right]_0^h = \left(a.\frac{h^3}{3}+b.\frac{h^2}{2}+c.h\right) \\ &= h.\frac{(4-9+6).y_i+(-8+12).y_{i+\frac{1}{2}}+(4-3).y_{i+1}}{6} = \frac{h}{6}.\left(y_i+4.y_{i+\frac{1}{2}}+y_{i+1}\right) \end{aligned}$$

d'où $\omega_0 = \omega_2 = \frac{1}{6}$ et $\omega_1 = \frac{2}{3}$ La méthode est de degré 3 et d'ordre 4. En effet :

$$\int_a^b f(x).dx \approx \frac{b-a}{n}.\sum_{i=0}^{n-1}\frac{y_i + 4.y_{i+\frac{1}{2}} + y_{i+1}}{6} \quad \Rightarrow \quad E_{rr}(f,n,2) \leq \frac{K}{n^4} \quad \text{avec} \quad K = \frac{\sup\left|f^{(4)}\right|}{180}.|b-a|^5$$

L'interpolation de Simpson est plus précise que l'interpolation trapèze lorsque la fonction à intégrer est raisonnablement continue. Elle se justifie beaucoup moins lorsque la fonction présente des discontinuités.

Lors d'une intégration "temps réel", elle introduit par ailleurs un décalage temporel de $2.h$ (pour éviter $y_{i+\frac{1}{2}}\ldots$) qui est souvent plus pénalisant que l'erreur d'intégration.

3.2.2.5 Autres méthodes

Méthode de	m	degré	poids	points
Boole-Villarceau	4	5	$\omega_0 = \omega_4 = \frac{7}{90}$; $\omega_1 = \omega_3 = \frac{16}{45}$; $\omega_2 = \frac{2}{15}$	$\lambda(i,j) = x_i + j.\frac{x_{i+1}-x_i}{4}$
Weddle-Hardy	6	7	$\omega_0 = \omega_6 = \frac{41}{840}$; $\omega_1 = \omega_5 = \frac{9}{35}$; $\omega_2 = \omega_4 = \frac{9}{280}$; $\omega_3 = \frac{34}{105}$	$\lambda(i,j) = x_i + j.\frac{x_{i+1}-x_i}{6}$

Voir aussi les méthodes de Newton-Cotes.

3.2.3 Avec le module scipy.integrate

Objectif : obtenir une approximation de $\int_a^b f(x).dx$

Pour obtenir une approximation d'une intégrale simple, double ou triple d'une fonction, le module `scipy.intergrate` propose les fonctions `quad`, `dblquad` et `tplquad`.

Ainsi `quad(f, a, b)` renvoie un tuple contenant une approximation de $\int_a^b f(x).dx$ et une estimation de l'erreur commise.

```
>>> scint.quad(lambda x : x**3, 0, 2)
(4.0, 4.440892098500626e-14)
```

$$\int_{yg}^{yd}\left(\int_{xg}^{xd} f(y,x).dx\right).dy \approx \texttt{scint.dblquad(f, xg, xd, lambda x: yg, lambda x, yd)}$$

```
def jac2(theta, phi):
    return m.sin(theta)

A = scint.dblquad(jac2, 0, 2*m.pi, lambda x: 0, lambda x: m.pi)
```

Aire d'une sphère de rayon unitaire.

Volume d'une boule de rayon unitaire.

```
def jac3(r, theta, phi):
    return r**2*m.sin(theta)

V = scint.tplquad(jac3, 0, 2*m.pi, lambda x: 0, lambda x: m.pi,
                  lambda x, y: 0, lambda x, y : 1)
```

Utiliser `cumtrapz` du module `scipy.integrate` permet d'intégrer une liste de valeurs, par exemple pour passer des mesures d'accélération à la vitesse en chaque point de mesure.

Ainsi `scipy.intergrate.cumtrapz(val, x=t, initial=i0)` permet d'obtenir l'évolution de l'aire sous la courbe formée par les éléments de `val`, avec `t`, la liste de leurs abscisses et `i0` la valeur initiale.

Remarque : la fonction ne semble pas tenir compte de la valeur initiale. Le troisième exemple serait peut-être la solution à adopter pour obtenir le résultat recherché :

```
>>> scint.cumtrapz([2, 3, 4], x=[0, 1, 2], initial=3)
array([3. , 2.5, 6. ])
>>> scint.cumtrapz([2, 3, 4], x=[0, 1, 2], initial=0)
array([0. , 2.5, 6. ])
>>> scint.cumtrapz([2, 3, 4], x=[0, 1, 2], initial=0) +3
array([3. , 5.5, 9. ])
```

3.3 Dérivation numérique

La dérivation numérique consiste à dériver (de façon approchée) une fonction sur un intervalle borné $[a, b]$, c'est-à dire calculer la pente de la courbe représentant la fonction, à partir d'un calcul ou d'une mesure en un nombre fini de points.

La répartition des points en abscisse est généralement uniforme (pas d'échantillonnage constant h) mais il existe des méthodes à pas variable, ou encore à pas adaptatif.

La dérivation des polynômes étant très simple, l'opération consiste généralement à construire une interpolation polynomiale par morceaux (de degré plus ou moins élevé) puis de dériver le polynôme sur chaque morceau.

La précision de l'intégration numérique peut s'améliorer en augmentant le nombre de points n (en diminuant le pas d'échantillonnage h) ou en augmentant le degré de l'interpolation polynomiale (sous réserve de bonnes propriétés de continuité de la courbe).

3.3.1 Dérivée première

3.3.1.1 Méthode à 1 pas

L'estimation de la dérivée la plus simple consiste à calculer la pente à partir du point courant et du point précédent ou suivant (Fig 3.10). L'estimation de la dérivée au point i peut alors se calculer par :

- différence avant : $D_i = \frac{1}{h}(y_{i+1} - y_i)$ (pente de la droite Δ^+),
- différence arrière : $D_i = \frac{1}{h}(y_i - y_{i-1})$ (pente de la droite Δ^-),

Évidement, lorsqu'il s'agit de dériver une fonction temporelle "en temps réel", le point suivant n'est pas encore connu donc seule la différence arrière peut être calculée.

Notons aussi que le calcul de la dérivée à partir de n points, conduit à un tableau de valeur de dimension $n-1$.

3.3.1.2 Méthode à 2 pas

Une méthode à 2 pas consiste à utiliser 3 points pour une meilleure estimation de la dérivée (Fig 3.11).

Sur un intervalle $[x_i, x_{i+2}]$ de longueur $H = 2.h$, la courbe est interpolée par un polynôme d'ordre 2 : $y = a_i.x^2 + b_i.x + c_i$. Les coefficients a_i, b_i et c_i ont déjà été calculés au paragraphe 3.2.2.4 en fonction des y_i. Il suffit

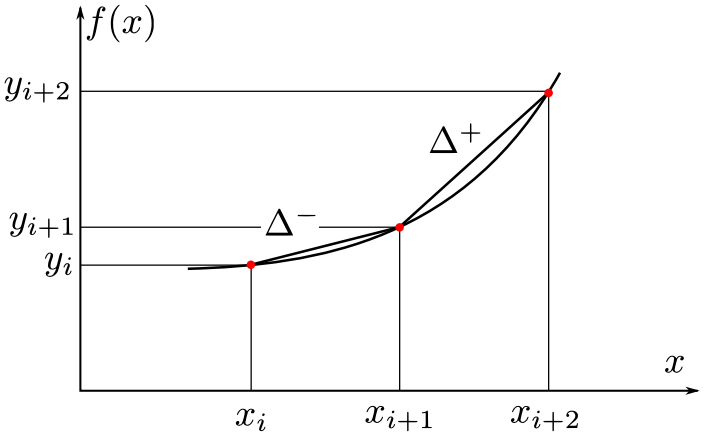


Figure 3.10 – Dérivation à 1 pas.

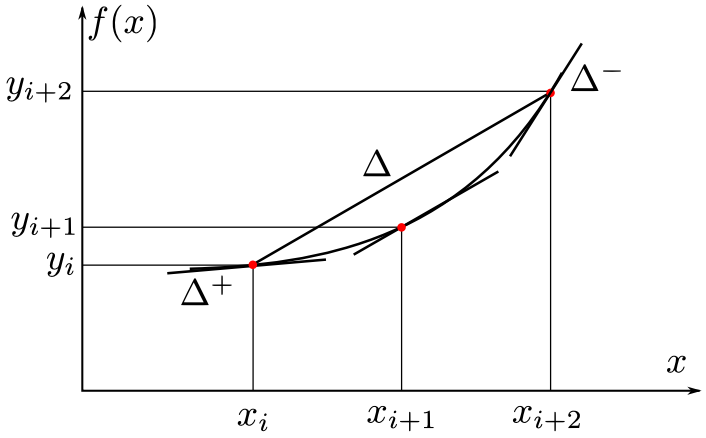


Figure 3.11 – Dérivation à 2 pas.

de dériver le polynôme respectivement en x_i, x_{i+1} et x_{i+2} pour obtenir respectivement les différences avant en x_i, centrée en x_{i+1} et arrière en x_{i+2}.

Après translation des abscisses, il s'agit en réalité de dériver en 0, $\frac{\mathrm{H}}{2} = h$ et $\mathrm{H} = 2.h$:

- différence avant :

$$\mathrm{D}_i = b_i = \frac{4.y_{i+1} - y_{i+2} - 3.y_i}{2.h} \text{ (pente de la droite } \Delta^+)$$

- différence centrée :

$$\begin{aligned}\mathrm{D}_{i+1} &= 2.a_i.h + b_i = 2.\frac{2.y_i + 2.y_{i+2} - 4.y_{i+1}}{4.h^2}.h + \frac{4.y_{i+1} - y_{i+2} - 3.y_i}{2.h} \\ &= \frac{y_{i+2} - y_i}{2.h} \text{ (pente de la droite } \Delta)\end{aligned}$$

- différence arrière :

$$\begin{aligned}\mathrm{D}_{i+2} &= 2.a_i.2.h + b_i = 4.\frac{2.y_i + 2.y_{i+2} - 4.y_{i+1}}{4.h^2}.h + \frac{4.y_{i+1} - y_{i+2} - 3.y_i}{2.h} \\ &= \frac{y_i - 4.y_{i+1} + 3.y_{i+2}}{2.h} \text{ (pente de la droite } \Delta^-)\end{aligned}$$

Il faut noter que la différence centrée est aussi simple à calculer que dans le cas d'une méthode à 1 pas, et correspond à la pente entre les deux points de part et d'autre du point courant. La précision étant d'ordre 2, elle constitue un très bon compromis.

Évidement, lorsqu'il s'agit de dériver une fonction temporelle "en temps réel", le ou les points suivants ne sont pas encore connus donc seule la différence arrière peut être calculée.

3.3.2 Dérivée seconde

Pour dériver deux fois une courbe donnée par une liste de points, le premier réflexe consiste à appliquer deux fois une dérivée simple (par une méthode à 1 ou 2 pas).

Il est cependant plus rapide de calculer directement la dérivée seconde à partir du polynôme d'interpolation, qui doit bien évidement être au moins de degré 2 pour obtenir un résultat non nul.

En utilisant une méthode à 2 pas, le polynôme d'interpolation s'écrit $y = a.x^2 + b.x + c$. Les coefficients a, b et c ont déjà été calculés au paragraphe 3.2.2.4 en fonction des y_i, ce qui conduit à une dérivée seconde constante sur tout l'intervalle $[x_i, x_{i+2}]$:

$$f''(x) = 2.a = 2.\frac{2.y_i + 2.y_{i+2} - 4.y_{i+1}}{H^2} = \frac{y_i + y_{i+2} - 2.y_{i+1}}{h^2}$$

Selon si cette valeur de la dérivée seconde est adoptée en i, $i+1$ ou $i+2$, il s'agit de la différence seconde avant, centrée ou arrière. On peut montrer que cela revient à calculer deux dérivées simples à 1 pas.

3.3.3 Influence du bruit de mesure

Lorsque la courbe est issue d'une mesure, elle est généralement entachée d'un léger bruit, qui peut devenir catastrophique pour l'évaluation de la dérivée (Fig 3.14).

En effet, si les points de mesure restent "en moyenne" au voisinage de la valeur mesurée, il existe des fluctuations rapides (c'est-à-dire à la fréquence d'échantillonnage) entre les points successifs (voir zoom de la Fig 3.14).

Le calcul de la dérivée conduit à déterminer la pente entre deux points successifs, ce qui perturbe fortement le signal dérivé et cache les évolutions lentes du signal (lentes devant la période d'échantillonnage).

Deux solutions sont possibles :

- filtrer (ou lisser) le signal d'origine pour supprimer l'essentiel du bruit, puis dériver,
- calculer la dérivée sur un temps plus long que le temps d'échantillonnage, par exemple pour une méthode à 1 pas en calculant la pente entre deux points espacés de k pas (solution adoptée pour $\dot{x}_L(t)$ sur la Fig 3.14, avec $k = 10$).

Dans les deux cas, le signal dérivé sera entaché d'un retard sur le signal d'origine, ce qui oblige à trouver un compromis entre la qualité du signal dérivé et le retard. k est généralement de l'ordre de 5 à 20 pas selon le bruit, le pas d'échantillonnage, les fréquences à observer dans le signal et le retard admissible.

Pour un lissage par moyenne mobile, on peut montrer que les deux méthodes s'avèrent identiques.

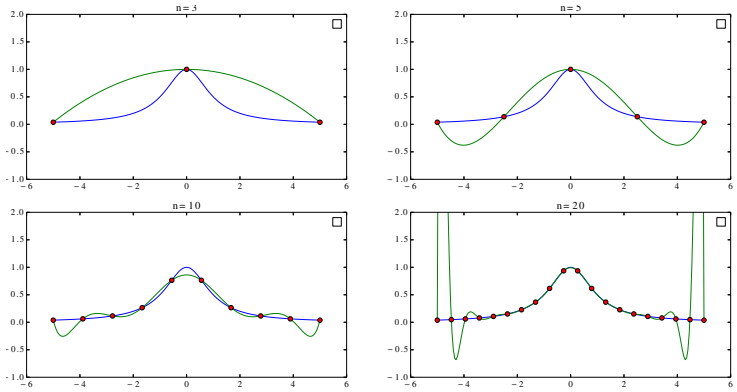


Figure 3.12 – Interpolation lagrangienne de degré n, $n \in \{3,5,10,20\}$, de la fonction $f(x) = \dfrac{1}{x^2+1}$.

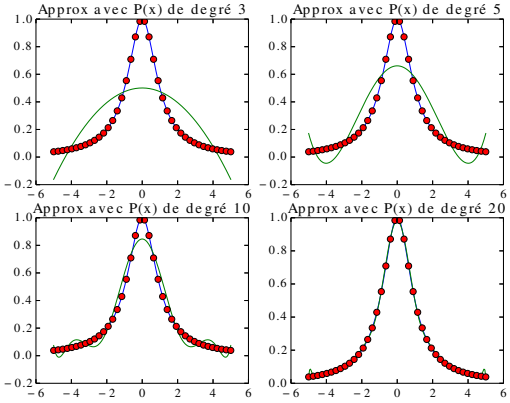


Figure 3.13 – Approximation avec un polynôme de degré n, $n \in \{3,5,10,20\}$, de la fonction $f(x) = \dfrac{1}{x^2+1}$ discrétisée en 40 points.

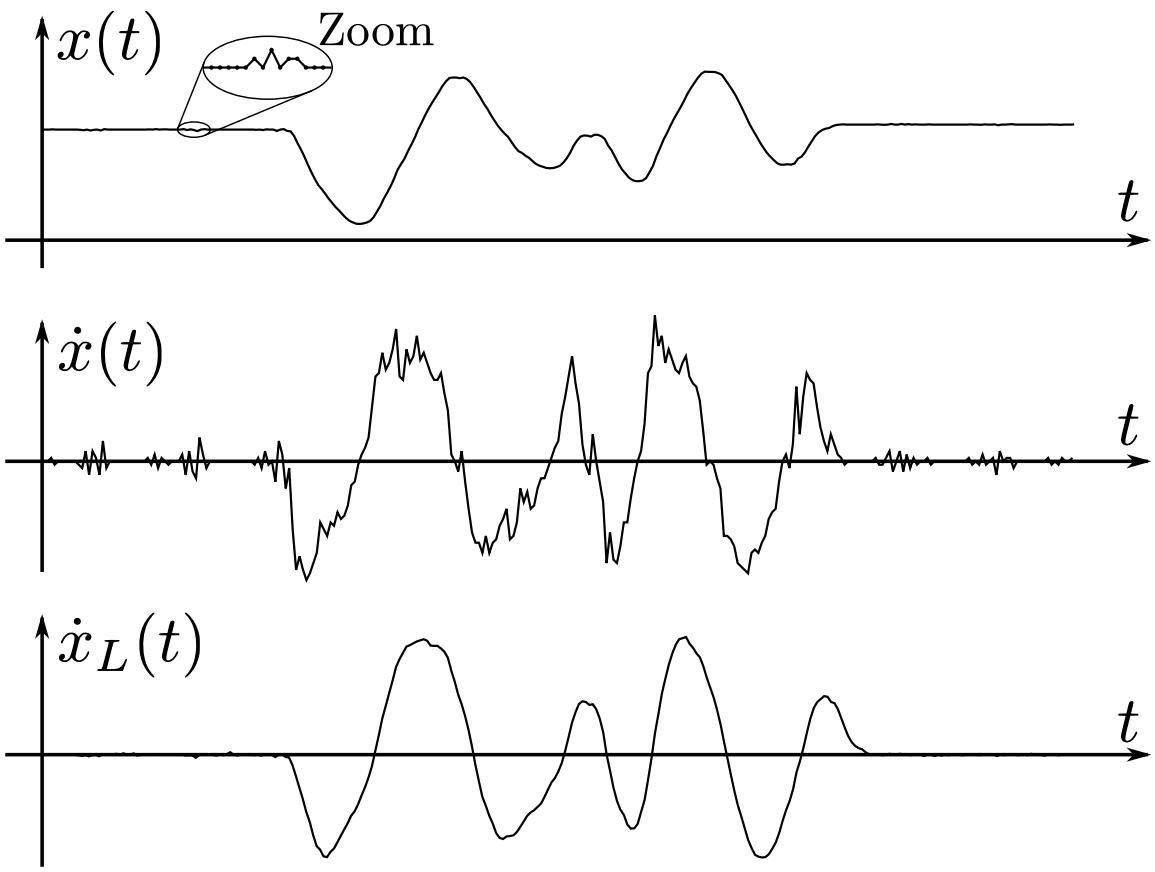


Figure 3.14 – Mesure d'une position au cours du temps $x(t)$ par un capteur potentiométrique, dérivée à 1 pas $\dot{x}(t)$ et dérivée à 1 pas lissée en effectuant la différence sur 10 pas.

Chapitre 4

Résolution approchée d'équations différentielles du premier ordre

Sommaire

4.1 Mise en place du problème

Une grande part des problèmes scientifiques se modélisent par une équation différentielle dont on cherche la solution pour dimensionner ou comprendre le phénomène.

Quand les équations sont simples (linéaires d'ordre 1 ou 2) la résolution analytique est aisée, mais nombre de modélisations conduisent à des équations non linéaires. Ce cours a pour objectif de proposer des méthodes pour déterminer une solution approchée.

4.1.1 Problème de Cauchy

Le problème de Cauchy consiste à trouver les fonctions $\mathbf{Y}$ de $[0,\mathrm{T}] \to \mathbb{R}^{\mathrm{N}}$, telles que :

$$\begin{cases} \dfrac{\mathrm{d}\mathbf{Y}}{\mathrm{dt}} = \mathbf{F}(\mathbf{Y},t) \\ \mathbf{Y}(t_0) = \mathbf{Y_0} \end{cases}$$

où $t_0 \in [0,\mathrm{T}]$ et $\mathbf{Y_0} \in \mathbb{R}^{\mathrm{N}}$ sont des données.

La plupart des systèmes d'équations différentielles de tout ordre peuvent se mettre sous cette forme de système d'équations différentielles du premier ordre [a] (un exemple sera traité plus loin).

a. à l'exception des équations différentielles implicites

4.1.2 Existence et unicité de la solution

Théorème de Cauchy-Lipschitz : soit $\mathbf{F}$ une fonction de $\mathbb{R}^{\mathrm{N}} \times [0,\mathrm{T}] \to \mathbb{R}^{\mathrm{N}}$ continue et lipschitzienne en $\mathbf{Y}$.

Alors, $\forall\ t_0 \in [0,\mathrm{T}]$ et $\forall\ \mathbf{Y_0} \in \mathbb{R}^{\mathrm{N}}$, le problème de Cauchy admet une solution unique définie sur $[0,\mathrm{T}]$.

Rappel : $\mathbf{F}$ lipschitzienne en $\mathbf{Y}$: $\exists\ k>0$ tel que,$\forall\ \mathbf{Y},\mathbf{Z} \in \mathbb{R}^{\mathrm{N}}, \forall\ t \in [0,\mathrm{T}]$, $\quad \|\mathbf{F}(\mathbf{Y},t) - \mathbf{F}(\mathbf{Z},t)\| \leq k.\|\mathbf{Y}-\mathbf{Z}\|$

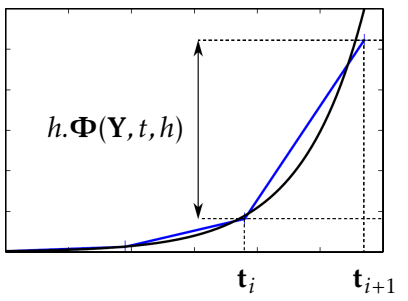
4.1.3 Résolution numérique

L'objectif est d'obtenir une solution approchée au problème de Cauchy pour une discrétisation temporelle de l'intervalle donné.

La démarche s'appuie sur la forme initiale de l'équation différentielle où il est facile de voir que la fonction $\mathbf{F}(t,\mathbf{Y})$ traduit l'évolution de $\mathbf{Y}$, c'est-à-dire la pente de la courbe $\mathbf{Y}(t)$. Les schémas d'intégration numériques exploitent $\mathbf{F}(\mathbf{Y},t)$ pour traduire l'évolution sur un pas de temps.

En notant h le pas de temps et N le nombre de pas de temps, l'intervalle est discrétisé par $t_0 = 0$, $t_1 = h$, ..., $t_{\mathrm{N}} = \mathrm{N}.h$. Sur un sous-intervalle donné, on cherche à déterminer la solution sous la forme :

$$\int_{t_i}^{t_{i+1}} \frac{\mathrm{d}\mathbf{Y}}{\mathrm{dt}}\mathrm{d}t = \int_{t_i}^{t_{i+1}} \mathbf{F}(\mathbf{Y},t)\mathrm{d}t \qquad \text{on en déduit que :} \mathbf{Y}_{i+1} = \mathbf{Y}_i + \int_{t_i}^{t_{i+1}} \mathbf{F}(\mathbf{Y},t)\mathrm{d}t$$



Les méthodes de résolution numérique des équations différentielles sont basées sur des techniques d'estimation de l'intégrale de la fonction $\mathbf{F}$.

On s'intéressera dans cette partie aux méthodes de résolution à un pas qui se mettent sous la forme générale ci contre où il faudra choisir $\mathbf{\Phi}$:

$$\begin{cases} \mathbf{Y}_{i+1} = \mathbf{Y}_i + h.\mathbf{\Phi}(\mathbf{Y},t,h) \\ \mathbf{Y}(t_0) = \mathbf{Y_0} \end{cases}$$

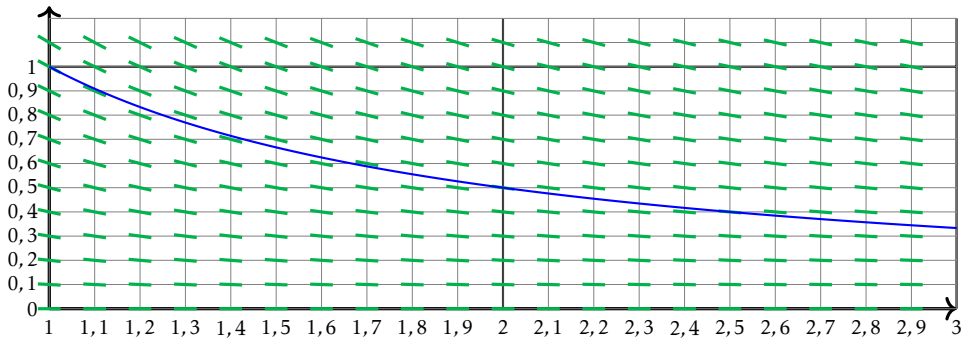
La fonction $\mathbf{\Phi}$ représente la pente de la droite permettant de passer de $\mathbf{Y}_i$ à $\mathbf{Y}_{i+1}$.

4.2 Méthodes à un pas

L'exemple élémentaire qui servira de support est la résolution de l'équation différentielle :

$$\frac{dy(t)}{dt} + \frac{y(t)}{t} = 0$$

avec $y(1) = 1$ pour $t \in [1,3]$.
La solution exacte est $y(t) = \frac{1}{t}$.

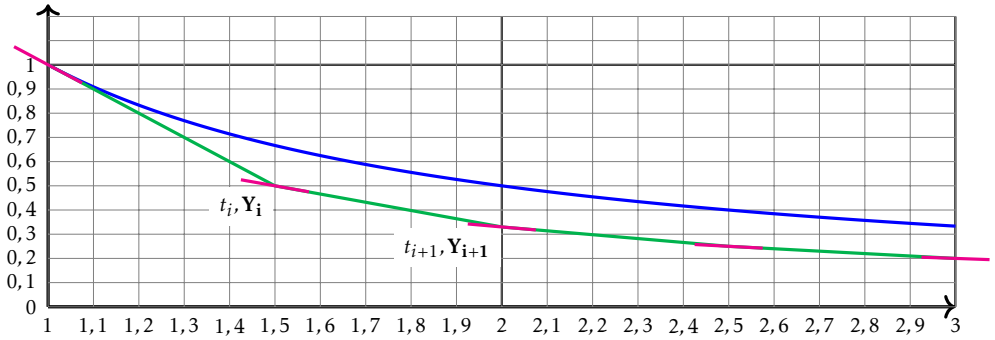


4.2.1 Méthode d'Euler explicite

4.2.1.1 Méthode

La première façon d'approximer l'intégrale est de réaliser une méthode des rectangles à gauche.
Ainsi $\int_{t_i}^{t_{i+1}} \mathbf{F}(\mathbf{Y},t)dt \approx h.\mathbf{F}(\mathbf{Y}_i,t_i)$.

On en déduit que $\mathbf{\Phi} = \mathbf{F}(\mathbf{Y}_i,t_i)$



Ainsi la relation de récurrence générale est :
$\mathbf{Y}_{i+1} = \mathbf{Y}_i + h.\mathbf{F}(\mathbf{Y}_i,t_i)$

La pente permettant de passer de $\mathbf{Y}_i$ à $\mathbf{Y}_{i+1}$ est la valeur de $\mathbf{F}(\mathbf{Y}_i,t_i)$ comme l'illustre la figure ci-dessus où le champ de la fonction $\mathbf{F}$ est tracée.

On peut aussi exprimer la relation sous la forme : $\mathbf{F}(\mathbf{Y}_i,t_i) = \frac{d\mathbf{Y}}{dt}(t_i) \simeq \frac{\mathbf{Y}_{i+1} - \mathbf{Y}_i}{h}$.

4.2.1.2 Propriétés (voir Annexe mathématique pour les définitions)

Stabilité

si $\mathbf{\Phi}$ est lipschitzienne alors le schéma sera stable.

Consistance

L'erreur de consistance est $c_i = \mathbf{Y}_{ex}(t_{i+1}) - \mathbf{Y}_{ex}(t_i) - h.\mathbf{F}(\mathbf{Y}_{ex}(t_i),t_i)$

Or le développement de Taylor-Lagrange donne : $\mathbf{Y}_{ex}(t_i+h) = \mathbf{Y}_{ex}(t_{i+1}) = \mathbf{Y}_{ex}(t_i) + h.\dot{\mathbf{Y}}_{ex}(t_i) + \frac{h^2}{2}.\ddot{\mathbf{Y}}_{ex}(\xi)$ avec $\xi \in [t_i, t_i+h]$.

Or $\dot{\mathbf{Y}}_{ex} = \mathbf{\Phi}(\mathbf{Y}_{ex}(t_i), t_i, h) = \mathbf{F}(\mathbf{Y}_{ex}(t_i), t_i)$

On en déduit que $c_i = \frac{h^2}{2}.\ddot{\mathbf{Y}}_{ex}(\xi)$ avec $\xi \in [t_i, t_i + h]$

Ainsi la méthode est consistante. Comme elle est stable, on en déduit qu'elle est convergente.

L'expression de l'erreur de consistance indique que la méthode est d'ordre 1.

4.2.1.3 Application

Q - 1 : *Écrire la relation de récurrence de la méthode d'Euler sur l'exemple.*

Q - 2 : *Montrer que pour certaines valeurs de h, la solution peut présenter des problèmes. Justifier et commenter les courbes de la Fig 4.1.*

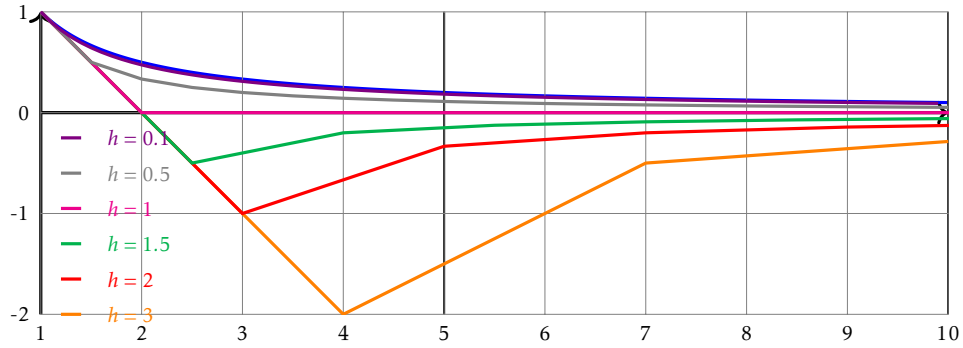


Figure 4.1 – Solution de l'application pour le schéma Euler explicite.

Le tableau suivant donne l'erreur comme étant le maximum de l'écart entre la solution exacte du problème (4.1) et la solution approchée ainsi que le temps de calcul pour différentes valeurs du pas de temps.

h	10^{-1}	10^{-2}	10^{-3}	10^{-4}	10^{-5}	10^{-6}
N pas de temps	100	1000	10000	100000	1000000	10000000
Erreur	$4.1.10^{-1}$	$4.9.10^{-2}$	$5.0.10^{-3}$	$5.0.10^{-04}$	$5.0.10^{-5}$	$5.0.10^{-06}$
Temps (s)	$5.2.10^{-3}$	$1.6.10^{-2}$	$7.8.10^{-2}$	$5.7.10^{-1}$	5.0	50

$$\begin{cases} y'(t) &= -y(t) \\ y(0) &= 1 \end{cases} \quad (4.1)$$

On constate que l'erreur évolue linéairement en fonction du pas de temps ainsi que le temps de calcul (complexité en O(N) car une seule boucle).

Il faut descendre à un pas de temps très faible avant d'obtenir un niveau d'erreur satisfaisant. On voit bien ici la limitation des méthodes d'ordre 1.

4.2.2 Méthode d'Euler implicite

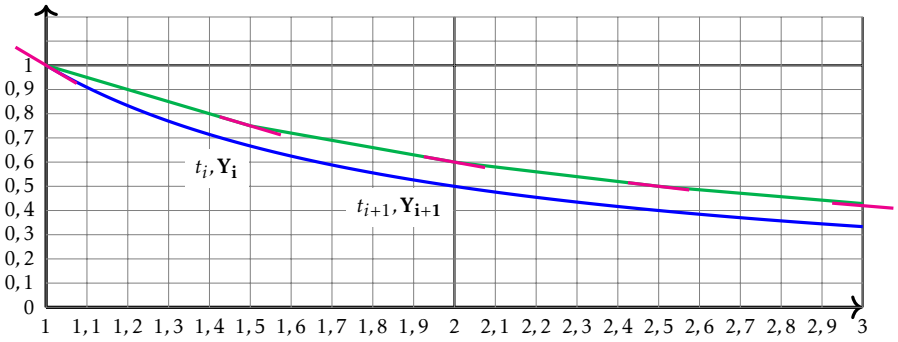
4.2.2.1 Méthode

La seconde façon d'approximer l'intégrale est de réaliser une méthode des rectangles à droite.

Ainsi :

$$\int_{t_i}^{t_{i+1}} \mathbf{F}(\mathbf{Y},t).\mathrm{d}t = h.\mathbf{F}(\mathbf{Y}_{i+1},t_{i+1})$$

On en déduit que $\mathbf{\Phi} = \mathbf{F}(\mathbf{Y}_{i+1},t_{i+1})$



Ainsi la relation de récurrence générale est : $\mathbf{Y}_{i+1} = \mathbf{Y}_i + h.\mathbf{F}(t_{i+1},\mathbf{Y}_{i+1})$

La pente permettant de passer de $\mathbf{Y}_i$ à $\mathbf{Y}_{i+1}$ est la valeur de $\mathbf{F}(\mathbf{Y}_{i+1},t_{i+1})$ comme l'illustre la figure ci-dessus où le champ de la fonction $\mathbf{F}$ est tracée.

On peut aussi exprimer la relation sous la forme : $\mathbf{F}(\mathbf{Y}_{i+1},t_{i+1}) = \dfrac{\mathrm{d}\mathbf{Y}}{\mathrm{d}t}(t_{i+1}) \simeq \dfrac{\mathbf{Y}_{i+1}-\mathbf{Y}_i}{h}$.

On remarque que le second membre dépend aussi de $\mathbf{Y}_{i+1}$. Il faut donc dans les cas non linéaires se ramener à un problème stationnaire à résoudre avec la méthode de Newton sur : $\mathbf{Y}_{i+1}-\mathbf{Y}_i-h.\mathbf{F}(\mathbf{Y}_{i+1},t_{i+1}) = 0$ ce qui alourdit notablement le calcul. Dans les cas linéaires, il faut au moins une inversion, ce qui est aussi très lourd pour les problèmes de grande dimension. Il faut donc retenir qu'une méthode implicite est généralement plus coûteuse qu'une méthode explicite à pas de temps égal.

4.2.2.2 Propriétés

De même que pour la méthode explicite, la méthode est convergente et d'ordre 1.

4.2.2.3 Application

Q - 3 : *Écrire la relation de récurrence de la méthode d'Euler implicite sur l'exemple.*

Q - 4 : *Montrer que quelle que soit la valeur de h, la solution ne peut plus présenter les mêmes problèmes qu'en explicite.*

La solution approchée pour différents pas de temps est donnée sur la Fig 4.2.

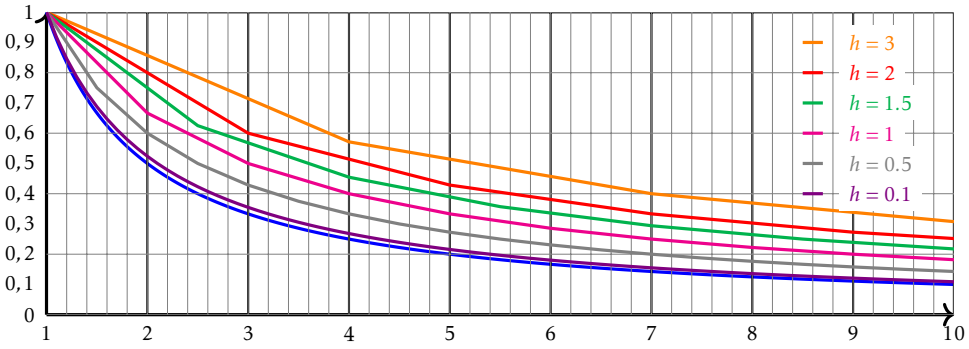


Figure 4.2 – Solution de l'application pour le schéma Euler implicite.

Le tableau suivant donne l'erreur comme étant le maximum de l'écart entre la solution exacte du problème (4.1) et la solution approchée ainsi que le temps de calcul pour différentes valeurs du pas de temps.

h	10^{-1}	10^{-2}	10^{-3}	10^{-4}	10^{-5}	10^{-6}
N pas de temps	100	1000	10000	100000	1000000	10000000
Erreur	$5.9.10^{-1}$	$5.0.10^{-2}$	$5.0.10^{-3}$	$5.0.10^{-04}$	$5.0.10^{-05}$	$5.0.10^{-06}$
Temps (s)	$5.2.10^{-3}$	$3.1.10^{-2}$	$3.7.10^{-1}$	3.7	37	370

On constate que l'erreur évolue linéairement en fonction du pas de temps ainsi que le temps de calcul (complexité en O(N) car une seule boucle). Le temps de calcul est cependant plus important à cause de la résolution de l'équation avec l'algorithme de Newton (on aurait pu inverser analytiquement la relation dans ce cas d'école).

Il faut descendre à un pas de temps très faible avant d'obtenir un niveau d'erreur satisfaisant. On voit bien ici la limitation des méthodes d'ordre 1.

4.2.3 Méthode de Runge-Kutta

Les méthodes d'Euler peuvent être vu comme une approximation du développement de Taylor-Lagrange à l'ordre 1. Pour augmenter la précision, une solution pourrait être d'augmenter l'ordre de développement ; c'est cependant impossible sans les expressions des dérivées n-ième de la fonction **F**.

Le schéma de Runge-Kutta permet d'obtenir des méthodes d'ordres plus élevés sans utiliser les fonctions dérivées de **F**.

4.2.3.1 Expression générale des méthodes de Runge-Kutta explicites

La méthode générale consiste à déterminer $\mathbf{\Phi}$ à partir d'approximations successives.

On choisit $\mathbf{\Phi}(\mathbf{Y},\tau,h) = \sum_{i=1}^{q} a_i.k_i(\mathbf{Y},\tau,h)$ avec
$k_1(\mathbf{Y},\tau,h) = \mathbf{F}(\mathbf{Y},\tau)$
pour $i \geq 2$, $\quad k_i(\mathbf{Y},\tau,h) = \mathbf{F}(\mathbf{Y} + \sum_{j=1}^{i-1} \beta_{ij}.k_j(\mathbf{Y},\tau,h), \tau + \alpha_i.h)$

On choisit les paramètres q, a_i, α_i et β_{ij} pour que la méthode soit d'ordre p.

On remarque que la méthode d'Euler explicite est le cas particulier de Runge-Kutta d'ordre 1.

En pratique, on va rarement au delà de l'ordre 4.

4.2.3.2 Runge-Kutta d'ordre 2 (RK2) – Méthode de Heun

Le relation de récurrence est $\mathbf{Y}_{i+1} = \mathbf{Y}_i + \frac{h}{2}.\mathbf{F}(\mathbf{Y}_i,t_i) + \frac{h}{2}.\mathbf{F}(\mathbf{Y}_i + h.\mathbf{F}(\mathbf{Y}_i,t_i), t_i + h)$.
On a donc pris $\mathbf{\Phi}(\mathbf{Y},\tau,h) = \frac{k_1+k_2}{2}$ avec :
$k_1 = \mathbf{F}(\mathbf{Y}_i,t_i)$ et $k_2 = \mathbf{F}(\mathbf{Y}_i + h.k_1, t_i + h)$.

Cette relation est une intégration de type trapèze entre la valeur au pas i et la valeur du pas $i+1$ estimée à partir d'une méthode d'Euler explicite.

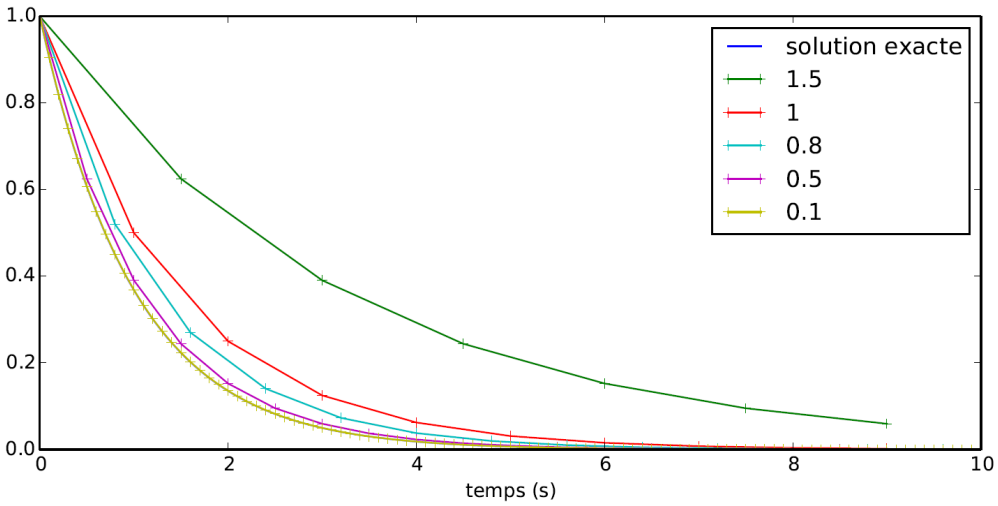


Figure 4.3 – Solution de l'application pour le schéma de Heun.

La solution approchée du problème (4.1) de l'application pour différents pas de temps est donnée sur la Fig 4.3.

Le tableau suivant donne l'erreur comme étant le maximum de l'écart entre la solution exacte et la solution approchée ainsi que le temps de calcul pour différentes valeurs du pas de temps.

h	10^{-1}	10^{-2}	10^{-3}	10^{-4}	10^{-5}	10^{-6}
N pas de temps	100	1000	10000	100000	1000000	10000000
Erreur	$1.8.10^{-2}$	$1.7.10^{-4}$	$1.7.10^{-6}$	$1.7.10^{-8}$	$1.7.10^{-10}$	$1.7.10^{-12}$
Temps (s)	$1.5.10^{-3}$	$1.4.10^{-2}$	$1.3.10^{-1}$	1.2	12	120

On constate que l'erreur évolue quadratiquement en fonction du pas de temps : en divisant le pas de temps par 10, l'erreur diminue d'un facteur 100.

Le temps de calcul est toujours linéaire car la complexité n'a pas augmenté (complexité en O(N)). Il est environ double de la méthode d'Euler explicite puisqu'il y a deux évaluations de $\mathbf{F}(\mathbf{Y}, t)$.

4.2.3.3 Runge-Kutta d'ordre 4 (RK4)

La fonction $\mathbf{\Phi}$ est $\mathbf{\Phi}(\mathbf{Y}, \tau, h) = \frac{1}{6}.(k_1 + 2k_2 + 2k_3 + k_4)$ avec quatre évaluations successives de $\mathbf{F}$:

$$
\begin{aligned}
k_1(\mathbf{Y}, \tau, h) &= \mathbf{F}(\mathbf{Y_i}, t_i) \\
k_2(\mathbf{Y}, \tau, h) &= \mathbf{F}\left(\mathbf{Y_i} + \frac{h}{2}.k_1(\mathbf{Y_i}, t_i, h), t_i + \frac{h}{2}\right) \\
k_3(\mathbf{Y}, \tau, h) &= \mathbf{F}\left(\mathbf{Y_i} + \frac{h}{2}.k_2(\mathbf{Y_i}, t_i, h), t_i + \frac{h}{2}\right) \\
k_4(\mathbf{Y}, \tau, h) &= \mathbf{F}\left(\mathbf{Y_i} + h.k_3(\mathbf{Y_i}, t_i, h), t_i + h\right)
\end{aligned}
$$

La solution approchée pour différents pas de temps est donnée sur la Fig 4.4.

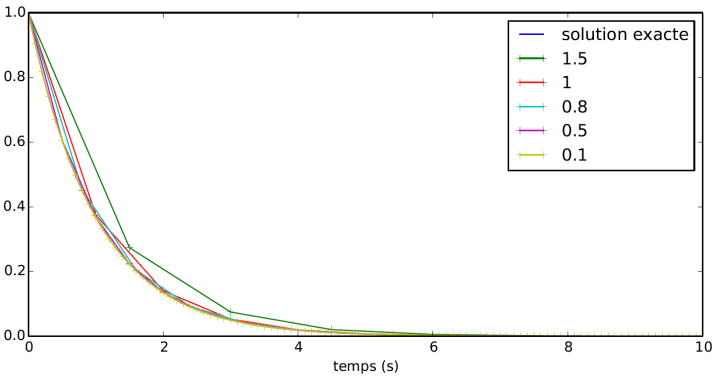


Figure 4.4 – Solution de l'application pour le schéma de Runge-Kutta d'ordre 4.

Le tableau suivant donne l'erreur comme étant le maximum de l'écart entre la solution exacte et la solution approchée ainsi que le temps de calcul pour différentes valeurs du pas de temps.

h	10^{-1}	10^{-2}	10^{-3}
N pas de temps	100	1000	10000
Erreur	$9.0.10^{-6}$	$8.4.10^{-10}$	$9.2.10^{-14}$
Temps (s)	$1.7.10^{-3}$	$1.6.10^{-2}$	$2.5.10^{-1}$

On constate que l'erreur évolue à l'ordre 4 en fonction du pas de temps : en divisant le pas de temps par 10, l'erreur diminue d'un facteur 10000.

Le temps de calcul est toujours linéaire car la complexité n'a pas augmentée (complexité en O(N)).

4.2.4 Conclusion

En terme de schéma explicite, la méthode de Runge-Kutta à l'ordre 4 est souvent utilisée.

Cependant, vous devez connaître uniquement la méthode d'Euler.

Concernant le choix du pas de temps, il obéit à un compromis entre le temps de calcul, la stabilité et le stockage en mémoire. Le choix de ce pas de temps dépendra essentiellement de la dynamique du système à représenter : si on veut représenter une dynamique à 10 kHz, il faudrait prendre un pas de temps de l'ordre de 10^{-5} (10 points par période).

Le choix du pas de temps est souvent fait de manière empirique par expérience de l'ingénieur.

4.2.5 Avec le module scipy.integrate

Le module `scipy.integrate` propose la fonction `odeint` telle que `odeint(F, Y0, T)` permet d'obtenir une estimation de la solution, pour chaque valeur de `T`, de l'équation différentielle $Y' = F(Y,t)$ en partant de `T[0]` avec comme valeur de l'état initial `Y0`. L'état du système peut contenir N variables avec $N \in \mathbb{N}^*$.

En ramenant l'équation différentielle scalaire du second ordre (4.2), à une équation différentielle vectorielle du premier ordre (4.3), le problème de Cauchy initial (4.4) peut être traité avec `odeint`.

$$\begin{cases} \frac{1}{\omega_0^2}.y''(t) + \frac{2.\xi}{\omega_0}.y'(t) + y(t) = K.(u(t-t_0) - u(t-t_1)) \\ y(t_0) = y_0 \text{ et } y'(t_0) = v_0 \end{cases} \quad (4.2)$$

avec $u(t)$ l'échelon unitaire, nul si t est négatif, égale à 1 sinon.

$$\begin{cases} \frac{dy(t)}{dt} &= v(t) \\ \frac{dv(t)}{dt} &= (K.(u(t-t_0) - u(t-t_1)) - y(t)).\omega_0^2 - 2.\xi.\omega_0.v(t) \\ y(t_0) &= y_0 \\ v(t_0) &= v_0 \end{cases} \quad (4.3)$$

$$\begin{cases} \frac{dY(t)}{dt} &= F(Y,t) \\ Y(t_0) &= [y_0, v_0] \end{cases} \quad (4.4)$$

avec $Y(t) = [y(t), v(t)]$

```python
def sys_amort(Y, t):
    if t < t0 or t > t1:
        u = 0
    else:
        u = 1
    return [Y[1], (K*u - Y[0]) * w0**2 - 2*xi*w0*Y[1]]
```

```python
plt.figure(0)
plt.title(r'Déplacement $y(t)$')
plt.plot(T, sim[:,0])
plt.figure(1)
plt.title(r'Vitesse $v(t)$')
plt.plot(T, sim[:,1])
```

```
K, xi, w0, t0, t1, y0, v0 = 3, 0.25, 20, 0, 1, 0, 0
T = np.linspace(t0-1, t1+1, 200)
sim = scint.odeint(sys_amort, [y0, v0], T)
```

- fonction associée au problème de Cauchy
- définition des paramètres et simulations
- présentation des résultats

4.3 Cas des équations différentielles linéaires d'ordre n à une dimension

4.3.1 Point de vu vectoriel

Nous avons traité jusqu'à présent les équations différentielles du premier ordre à une dimension : $\frac{\mathrm{d}y}{\mathrm{d}t} = f(y,t)$.

Pour étendre cette étude au cas des équations différentielles linéaires d'ordre n à une dimension, introduisons alors l'équation suivante :

$$a_n.(t).\frac{\mathrm{d}^n y}{\mathrm{d}t^n}(t) + \ldots + a_1(t).\frac{\mathrm{d}y}{\mathrm{d}t}(t) + a_0(t).y(t) = f_a(t) \quad \Rightarrow \quad \sum_{i=0}^{n} a_i(t).\frac{\mathrm{d}^i y}{\mathrm{d}t^i}(t) = f_a(t) \tag{4.5}$$

Si l'équation différentielle est d'ordre n alors $a_n(t)$ est différents de 0 et $\forall m \in \mathbb{N}/m > n,\ a_m(t) = 0$. Posons alors,

$$\forall i \in [\![0, n-1]\!],\ b_i = \frac{a_i}{a_n} \quad \text{et} \quad f_b(t) = \frac{1}{a_n(t)}.f_a(t)$$

et construisons le vecteur $\vec{\mathrm{Y}}(t)$ de dimension n tel que :

$$\vec{\mathrm{Y}}(t) = \begin{bmatrix} y(t) \\ \frac{\mathrm{d}y}{\mathrm{d}t}(t) \\ \vdots \\ \frac{\mathrm{d}^{n-1}y}{\mathrm{d}t^{n-1}}(t) \end{bmatrix} \quad \Rightarrow \quad \left[\vec{\mathrm{Y}}(t)\right]_i = \frac{\mathrm{d}^{i-1}y(t)}{\mathrm{d}t^{i-1}}$$

Alors (4.5) se ramène à l'équation suivante :

$$\begin{bmatrix} \frac{\mathrm{d}y}{\mathrm{d}t}(t) \\ \frac{\mathrm{d}^2y}{\mathrm{d}t^2}(t) \\ \vdots \\ \frac{\mathrm{d}^{n-1}y}{\mathrm{d}t^{n-1}}(t) \\ \frac{\mathrm{d}^n y}{\mathrm{d}t^n}(t) \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & \ldots & 0 \\ 0 & 0 & 1 & \ldots & \vdots \\ \vdots & & & \ddots & \vdots \\ 0 & & & & 1 \\ -b_0 & -b_1 & -b_2 & \ldots & b_{n-1} \end{bmatrix} \begin{bmatrix} y(t) \\ \frac{\mathrm{d}y}{\mathrm{d}t}(t) \\ \vdots \\ \frac{\mathrm{d}^{n-2}y}{\mathrm{d}t^{n-2}}(t) \\ \frac{\mathrm{d}^{n-1}y}{\mathrm{d}t^{n-1}}(t) \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ f_b(t) \end{bmatrix} \quad \Rightarrow \quad \frac{\mathrm{d}\vec{\mathrm{Y}}}{\mathrm{d}t}(t) = \underline{\underline{\mathrm{A}}}(t).\vec{\mathrm{Y}}(t) + \mathrm{B}(t) = \underline{\underline{f}}\left(\vec{\mathrm{Y}}(t), t\right)$$

ce qui correspond à l'étude des parties précédentes mais cette fois en dimension n.

4.3.2 Mise en forme des systèmes d'équations différentielles

Nous avons dit au départ que tout système d'équations différentielles pouvait se mettre sous la forme du problème de Cauchy. Traitons quelques exemples.

4.3.2.1 Equation harmonique

Prenons un système masse-ressort. Son équation du mouvement se met sous la forme :
$\ddot{y}(t) + \omega_0^2.y(t) = 0$. Les conditions initiales sont $y(0) = 1$ et $\dot{y}(0) = 0$.

Q - 5 : *Mettre ce problème sous la forme du problème de Cauchy.*

L'idée est d'introduire des variables supplémentaires. Ici, on introduit la fonction $y_2(t)$ telle qu'elle soit solution de $y_2(t) = \dot{y_1}(t)$ avec $y_1(t) = y(t)$.

L'équation différentielle du système masse-ressort s'écrit ainsi $\dot{y_2}(t) = -\omega_0^2.y_1(t)$.

En posant $\mathbf{Y}(t) = \begin{pmatrix} y_1(t) \\ y_2(t) \end{pmatrix}$, $\mathbf{F}(\mathbf{Y}, t) = \begin{pmatrix} y_2(t) \\ -\omega_0^2 y_1(t) \end{pmatrix}$ et $\mathbf{Y}_0 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$, on obtient bien la forme du problème de Cauchy.
La résolution se fait de la même manière que précédemment sauf que cette fois les grandeurs manipulées sont des vecteurs et non plus des scalaires.

Une autre solution classique permettant de résoudre cette équation différentielle avec la méthode d'Euler explicite est d'appliquer deux fois la définition.

On note Yp la dérivée première. On a $Yp_i = \dfrac{Y_{i+1} - Y_i}{h}$

On note Ypp la dérivée seconde. On a $Ypp_i = \dfrac{Yp_{i+1} - Yp_i}{h} = \dfrac{Y_{i+2} - 2.Y_{i+1} + Y_i}{h^2}$.

On en déduit que $Y_{i+2} = (-h^2.\omega_0^2 - 1).Y_i + 2.Y_{i+1}$. On obtient une relation de récurrence directe qui peut être programmée.

Ce schéma est un schéma d'intégration à deux pas qui pourrait s'inscrire dans un cadre plus général des méthodes à plusieurs pas.

4.3.2.2 Système masse-ressort-amortisseur entretenu

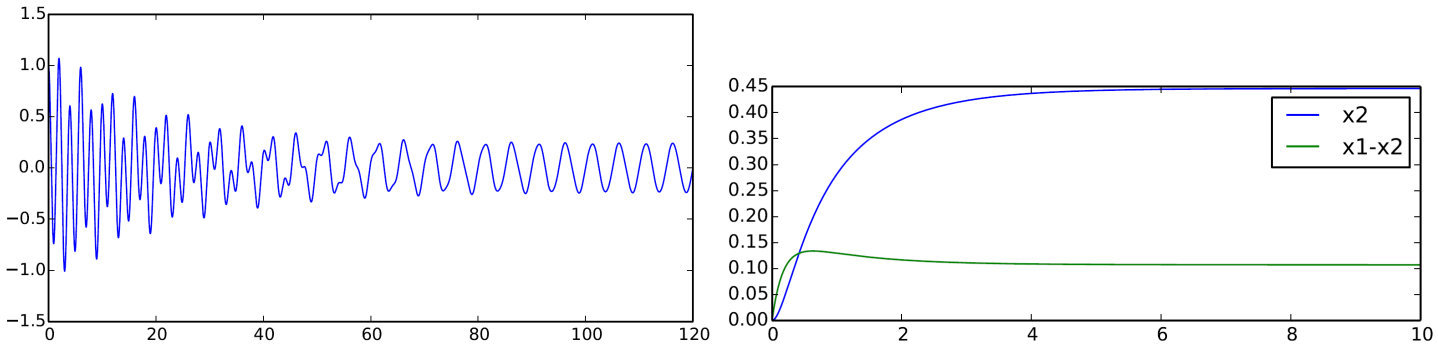
Soit un système masse-ressort-amortisseur soumis à une excitation extérieure harmonique.

L'équation différentielle se met sous la forme générale : $\ddot{y}(t) + 2.\xi.\omega_0.\dot{y}(t) + \omega_0^2.y(t) = f_0.\sin(\omega t)$. Les conditions initiales sont $y(0) = 1$ et $\dot{y}(0) = 0$ (solution Fig 4.5(a)).

Q - 6 : *Mettre ce problème sous la forme du problème de Cauchy.*

4.3.2.3 Création du glycol

Le glycol (noté E) résulte de l'addition d'eau à l'oxyde d'éthylène (noté O) en phase gazeuse, selon la réaction $O + H_2O \xrightarrow{(1)} E$ de constante de vitesse k_1. Cette réaction est effectuée à 473 K sous une pression P = 15,0 bar.



(a) Solution de l'équation différentielle du système masse-ressort- amortisseur pour $\omega_0 = 2$ rad.s^{-1}, $\xi = 0.015$, $f_0 = 2$ N et $\omega = 5$ rad.s^{-1}.

(b) Solution de l'équation différentielle de la réaction de création du glycol.

Figure 4.5 – Solutions des exemples 2 et 3.

Industriellement le temps de passage dans le réacteur ne permet pas d'atteindre l'état d'équilibre thermodynamique et on constate l'apparition de diéthylèneglycol (noté D) produit par la réaction $O + E \xrightarrow{(2)} D$ se déroulant également en phase gazeuse, de constante de vitesse k_2. Les réactions sont supposées d'ordre un par rapport à chacun des réactifs et totales. Pour traduire le fait que l'eau réagit moins vite que le glycol E sur l'oxyde d'éthylène O, les constantes de vitesse k_1 et k_2 sont choisies telles que $k_2 = 5k_1$.

Le mélange initial est constitué d'oxyde d'éthylène et d'eau à la concentration molaire $c_0 = 1{,}00$ mol.L^{-1} chacun. On notera x_1 l'avancement de la réaction (1) en mol.L^{-1} et x_2 l'avancement de la réaction (2) en mol.L^{-1}.

Les équations différentielles modélisant le mécanisme réactionnel sont (solution Fig 4.5(b)) :

Q - 7 : *Mettre ce problème sous la forme du problème de Cauchy.*

$$\begin{cases} \dfrac{dx_1}{dt} = k_1.(c_0 - x_1 - x_2).(c_0 - x_1) \\ \dfrac{dx_2}{dt} = k_2.(c_0 - x_1 - x_2).(x_1 - x_2) \end{cases}$$

4.4 Annexe mathématique

4.4.1 Définitions

On note $\mathbf{Y}_{ex}(t)$ la solution exacte.

4.4.1.1 Erreur de consistance au pas i

Définition : Erreur de consistance au pas i

On note c_i l'erreur de consistance au pas i telle que $c_i = \mathbf{Y}_{ex}(t_{i+1}) - \mathbf{Y}_{ex}(t_i) - h.\mathbf{\Phi}(\mathbf{Y}_{ex}(t_j), t_j, h)$

Elle permet de définir l'erreur réalisée entre l'approximation et la solution exacte au pas de temps i.

4.4.1.2 Méthode consistante

Définition : Méthode consistante

La méthode est dite consistante si $\lim_{h \to 0} \sum_{i=1}^{N} |c_i| = 0$

4.4.1.3 Méthode stable

Soit le schéma choisi et le schéma perturbé

$$\begin{cases} \mathbf{Y}_{i+1} = \mathbf{Y}_i + h.\mathbf{\Phi}(\mathbf{Y}_j, t_j, h) \\ \mathbf{Y}_{ex}(t_0) = \mathbf{Y_0} \end{cases} \qquad \begin{cases} \mathbf{Y}^*_{i+1} = \mathbf{Y}^*_i + h.\mathbf{\Phi}(\mathbf{Y}^*_j, t_j, h) + \zeta_i \\ \mathbf{Y}^*_{ex}(t_0) = \mathbf{Y}^*_0 \end{cases}$$

Définition : Méthode stable

La méthode est stable si $\exists\ \mathrm{M} > 0 / \sup_{0 \le i \le \mathrm{N}} |\mathbf{Y}^*_i - \mathbf{Y}_i| \le \mathrm{M}.\left(|\mathbf{Y}^*_0 - \mathbf{Y_0}| + \sum_{i=0}^{\mathrm{N}} |\zeta_i|\right)$

4.4.1.4 Méthode convergente

Définition : Méthode convergente

La méthode est convergente si

$$\lim_{h \to 0}\left[\sup_{0 \le i \le \mathrm{N}} |\mathbf{Y}_{ex}(t_i) - \mathbf{Y}_i|\right] = 0$$

4.4.1.5 Ordre de convergence

Définition : Méthode d'ordre p

Une méthode est dite d'ordre p si pour h au voisinage de 0 :

$$\exists\ \mathrm{M} > 0 / \forall\ i, |c_i| \le \mathrm{M}.h^{p+1}$$

4.4.2 Propriétés

4.4.2.1 Propriété 1

Une méthode est convergente si et seulement si elle est stable et consistante.

4.4.2.2 Propriété 2

Si $\mathbf{\Phi}$ est lipschitzienne en $\mathbf{Y}$ alors le schéma numérique est stable.

Démonstration :
Soit le schéma numérique et le schéma perturbé avec $\mathbf{\Phi}$ $k-$lipschitzienne.

On a : $|\mathbf{Y}^*_i - \mathbf{Y}_i| = |\mathbf{Y}^*_{i-1} - \mathbf{Y}_{i-1} + h.(\mathbf{\Phi}(\mathbf{Y^*}_j, t_j, h) - \mathbf{\Phi}(\mathbf{Y}_j, t_j, h)) + \zeta_{i-1}|$

Or $\mathbf{\Phi}$ est $k-$lipschitzienne et par inégalité triangulaire, on obtient

$|\mathbf{Y}^*_i - \mathbf{Y}_i| \le (1 + h.k).|\mathbf{Y}^*_{i-1} - \mathbf{Y}_{i-1}| + |\zeta_{i-1}|$

Par récurrence simple, $|\mathbf{Y}^*_i - \mathbf{Y}_i| \le (1 + h.k)^i.|\mathbf{Y}^*_0 - \mathbf{Y}_0| + \sum_{k=0}^{i-1} |\zeta_k|$

Comme $1 + h.k > 1$, $1 \le (1 + h.k)^i \le (1 + h.k)^{\mathrm{N}}$, ainsi

$\forall\ i, \qquad |\mathbf{Y}^*_i - \mathbf{Y}_i| \le \mathrm{M}.\left(|\mathbf{Y}^*_0 - \mathbf{Y_0}| + \sum_{i=0}^{\mathrm{N}} |\zeta_i|\right)$

Donc la borne supérieure est également majorée par ce terme, on en déduit que le schéma est stable.

Chapitre 5

Pivot de Gauss - Résolution d'un système de Cramer

$$\begin{cases} a_{11}.x_1 + a_{12}.x_2 + \cdots + a_{1\mathrm{n}}.x_\mathrm{n} &= b_1 \\ a_{21}.x_1 + a_{22}.x_2 + \cdots + a_{2\mathrm{n}}.x_\mathrm{n} &= b_2 \\ \vdots &= \vdots \\ a_{n1}.x_1 + a_{n2}.x_2 + \cdots + a_{\mathrm{nn}}.x_\mathrm{n} &= b_\mathrm{n} \end{cases} \Rightarrow \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1\mathrm{n}} \\ a_{21} & a_{22} & \cdots & a_{11} \\ \vdots & \vdots & \vdots & \vdots \\ a_{\mathrm{n}1} & a_{\mathrm{n}2} & \cdots & a_{\mathrm{nn}} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_\mathrm{n} \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_\mathrm{n} \end{bmatrix}$$

Sommaire

5.1 Introduction

5.1.1 Système d'équations linéaires

Dans les domaines du calcul de structure, de la simulation multiphysique ou encore en métrologie, il est fréquent d'avoir à résoudre des systèmes d'équations linéaires. L'algorithme de Gauss permet de résoudre ces systèmes. Soit un système de n équations linéaires, on peut l'écrire sous la forme matricielle A.X = B :

$$\begin{cases} a_{11}.x_1 + a_{12}.x_2 + \cdots + a_{1n}.x_n &= b_1 \\ a_{21}.x_1 + a_{22}.x_2 + \cdots + a_{2n}.x_n &= b_2 \\ \vdots &= \vdots \\ a_{n1}.x_1 + a_{n2}.x_2 + \cdots + a_{nn}.x_n &= b_n \end{cases} \Rightarrow \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{11} \\ \vdots & \vdots & \vdots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

5.1.2 Méthode directe

En utilisant la méthode de Gauss, l'objectif est de résoudre ce système en *une fois*. Ce type de méthode est appelé directe en opposition aux méthodes itératives pour lesquelles on répète des opérations jusqu'à ce que le résultat converge vers une solution.

Les méthodes directes sont encore majoritairement employées dans les codes actuels car elles sont robustes et que l'on est capable de déterminer à l'avance leur coût (nombre d'opérations, stockage nécessaire). Cependant, le coût pour la méthode de Gauss augmente fortement pour des systèmes de grande taille.

5.1.3 Quelques facteurs influents

Les systèmes auxquels on va s'intéresser dans la suite sont supposés linéaires et inversibles.

Si la dimension de la matrice dim(A) = n est importante le nombre de données à stocker est très élevé (n^2). Afin de réduire les ressources mémoire nécessaires, le stockage des données devra être optimisé.

De plus, lors du stockage numérique de ces données, chaque nombre est codé sur un nombre fini de bits. Selon le principe de la *virgule flottante*, des approximations sont donc commises. Or pour les systèmes de grande taille, ou mal conditionnés (5.4.2), la méthode de Gauss a tendance à amplifier les erreurs d'arrondi (d'où une instabilité numérique).

5.2 Méthode de Gauss

5.2.1 Principe du pivot

5.2.1.1 Transvection

Pour appliquer la méthode de Gauss, on procède par opérations élémentaires sur les lignes L_i d'un système d'équations. Or la solution d'un système linéaire ne change pas si :

- on multiplie tous les termes d'une équation par une constante non nulle :

$$L_i \leftarrow \lambda.L_i \quad \text{avec} \quad \lambda \neq 0$$

- on remplace une équation par la somme, membre à membre de cette équation et d'une autre équation du système :

$$L_i \leftarrow L_i + \lambda.L_j$$

On construit alors la matrice de transvection $T_{i,j}(\lambda)$, de dimension n, définie par :

$$[T_{i,j}(\lambda)]_{k,l} = \begin{cases} 1 & \text{si } k = l \\ \lambda & \text{si } k = i \text{ et } l = j \\ 0 & \text{sinon} \end{cases} \quad \Leftrightarrow \quad T_{i,j}(\lambda) = \begin{matrix} \\ L_i \\ \\ \\ \\ \end{matrix} \begin{matrix} \begin{matrix} & & C_j & & \end{matrix} \\ \begin{bmatrix} 1 & & 0 & & \\ & 1 & \lambda & & \\ & & \cdot & & \\ & 0 & 1 & & \\ & & & & 1 \end{bmatrix} \end{matrix} = \mathbb{I}_n + \lambda.E_{i,j}$$

avec $\mathbb{I}_n$, la matrice identité de dimension n et $E_{i,j}$ la matrice constituée de 0 sauf ligne i et colonne j où elle admet 1 comme cœfficient.

5.2.1.2 Triangularisation

L'idée principale de la méthode de Gauss est d'obtenir, par opérations élémentaires, un système triangulaire d'équations, où la ligne L_i s'exprime en fonction de i (resp. $n-i+1$) inconnues dont une seule n'est pas apparue (resp. n'apparaît pas) dans les lignes précédentes (resp. suivantes) :

$$\begin{cases} a'_{11}.x_1 + a'_{12}.x_2 + \cdots + a'_{1n}.x_n &= b'_1 \\ a'_{22}.x_2 + \cdots + a'_{2n}.x_n &= b'_2 \\ \ddots \quad \vdots &= \vdots \\ a'_{nn}.x_n &= b'_n \end{cases} \Rightarrow \begin{bmatrix} a'_{11} & a'_{12} & \cdots & a'_{1n} \\ & a'_{22} & \cdots & a'_{11} \\ & & \ddots & \vdots \\ & & & a'_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b'_1 \\ b'_2 \\ \vdots \\ b'_n \end{bmatrix}$$

Pour trianguliariser, il faut éliminer des cœfficients sous la diagonale de la matrice. A l'étape i :

$$\forall\, k \in [\![i+1, n]\!],\ L_k \leftarrow L_k - \frac{a_{k,i}}{a_{i,i}}.L_i$$

On utilise donc des pivots, les cœfficients $a_{i,i}$.

Il apparaît donc un problème mathématiquement si $a_{i,i} = 0$ et numériquement si $a_{i,i} \ll 1$.

Pour éviter ce problème deux solutions existent :

- méthode du pivot partiel : permutation des lignes ou des colonnes $\Rightarrow$ méthode simple
- méthode du pivot total : permutation des lignes et des colonnes $\Rightarrow$ méthode plus robuste

Dans le cas du pivot partiel sur les lignes :

Algorithm 3 Pivot partiel sur les lignes

entrée: A,B deux tableaux
pour i de 1 à $n-1$ **faire**
  déterminer j tel que $a_{j,i} = \sup_{i \le k \le n} |a_{k,i}|$
  inverser ligne i et ligne j sur A et sur B
  $L_i, L_j \leftarrow L_j, L_i$
  pour k de $i+1$ à n **faire**
    $L_k \leftarrow L_k - \frac{a_{k,i}}{a_{i,i}}.L_i$
  fin pour
fin pourrenvoi: A,B

5.2.1.3 Remontée

Une fois la matrice triangularisée, si le problème initial est un système de Cramer, on obtient simplement la solution, ligne par ligne :

$$\forall i \in [\![1,n]\!]\ x_i = \frac{1}{a'_{i,j}}.\left(b'_i - \sum_{k=i+1}^{n} a'_{i,k}.x_k\right)$$

Algorithm 4 Remontée

entrée: A',B'
pour i de n à 1 **faire**
  pour j de $i+1$ à n **faire**
    $b'_i = b'_i - a'_{i,k}.x_k$
  fin pour
  $x_i = \frac{b'_i}{a'_{i,i}}$
fin pour
renvoi: x

5.2.2 Inversion de matrice

5.2.2.1 Balayage

Lorsque la matrice est triangularisée, on utilise une méthode identique à celle de la triangularisation pour diagonaliser la matrice A.

```
for k in range(N-1,0,-1):
        for i in range(k):
            b[i] -= b[k]*A[i,k]/A[k,k]
            A[i,:] -= A[k,:]*A[i,k]/A[k,k]
```

Algorithm 5 Diagonalisation

entrée: A',B'
pour i de n à 2 **faire**
  pour k de $i-1$ à 1 **faire**
    $L_k \leftarrow L_k - \frac{a'_{k,i}}{a'_{i,i}}.L_i$
  fin pour
fin pour
renvoi: A',B'

5.2.2.2 Résolution

Le système d'équation ayant été diagonalisé, le problème de Cramer initial peut alors s'écrire :

$$\begin{cases} a''_{11}.x_1 &= b''_1 \\ a''_{22}.x_2 &= b''_2 \\ \vdots &= \vdots \\ a''_{nn}.x_n &= b''_n \end{cases} \Rightarrow \begin{bmatrix} a''_{11} & 0 & \cdots & 0 \\ 0 & a''_{22} & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \dots & 0 & a''_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b''_1 \\ b''_2 \\ \vdots \\ b''_n \end{bmatrix} \Rightarrow \quad \forall x \in [\![1,n]\!]\ x_i = \frac{b''_i}{a''_{i,i}}$$

Algorithm 6 Résolution

entrée: A",B"
pour i de 1 à n **faire**
  $x_i = \frac{b''_i}{a''_{i,i}}$
fin pour
renvoi: x

5.2.2.3 Matrice inverse

On suppose la matrice A inversible, donnant une solution unique au problème de Cramer A.X = B. On appelle alors :

- $(T_i)_1^N$ les N matrices de transvection permettant de triangulariser la matrice A (passer de A à A') :

$$\underbrace{T_N.T_{N-1}.\dots.T_1}_{T}.A = A'$$

- $(S_i)_1^M$ les M matrices de transvection permettant de diagonaliser A' (passer de A' à A")

$$\underbrace{S_M.S_{M-1}.\dots.S_1}_{S}.A' = A$$

- D, la matrice diagonale ayant pour cœfficient diagonaux $d_{i,i} = \frac{1}{a''_{i,j}}$

On obtient donc D.S.T.A = I_n. La matrice A étant inversible, D.S.T apparaît donc comme l'inverse de A.

Ainsi pour déterminer l'inverse d'une matrice inversible A, il suffit d'appliquer à la matrice identité les mêmes opérations élémentaires sur les lignes ou les colonnes qui permettent de passe de A à I_n.

5.3 Avec le module numpy.linalg

Pour faire de l'algèbre linéaire avec `Python`, le plus simple peut être d'utiliser le module `numpy.linalg` contenant les méthodes `det(A)` pour le calcul du déterminant de `A`, `inv(A)` pour le calcul de la matrice inverse de `A` et `solve(A,B)` pour obtenir la solution du problème linéaire A.X = B (où X est l'inconnue).

La classe `array` (tableaux `numpy`) possède beaucoup de méthodes notamment `T` pour obtenir la transposée d'un tableau et `dot(B)` pour multiplier un tableau à droite par `B`.

```
A = np.array([[2, 1, -1], [6, -1, -2], [-6, -2, 3]])
## Vérification du déterminant
print(nalg.det(A))
X = np.array([2, 3, 7]).T
## Produit matriciel de A par B
B = A.dot(X)
```

```
## Résolution par calcul
## de la matrice inverse
inv_A = nalg.inv(A)
X_sol_1 = inv_A.dot(B)
## Résolution avec solve
X_sol_2 = nalg.solve(A,B)
```

5.4 Approximations

5.4.1 Sources d'approximation dans le calcul numérique

Les deux sources d'erreur qui interviennent systématiquement dans le calcul numérique sont :

- les erreurs de troncature ou de discrétisation qui proviennent de simplifications du modèle mathématique comme par exemple le remplacement d'une dérivée par une différence finie, le développement en série de Taylor limité, etc.
- les erreurs d'arrondi qui proviennent du fait qu'il n'est pas possible de représenter (tous) les réels exactement dans un ordinateur.

Rappel
Dans un ordinateur, une variable réelle $x \neq 0$ est représentée en virgule flottante par $x = (-1)^s.m.b^e$ où m est la mantisse (ou partie fractionnelle), b la base (2 classiquement dans un ordinateur) et e est l'exposant.

La répartition suivant la norme IEEE pour coder un nombre en virgule flottante est :

Simple précision (32 bits)	1 bit pour le signe, 8 bits pour l'exposant et 23 bits pour la mantisse
Double précision (64 bits)	1 bit pour le signe, 11 bits pour l'exposant et 52 bits pour la mantisse

La précision en virgule flottante est définie comme le plus petit nombre positif ε tel que $float(1+\varepsilon) > 1$.

On peut montrer que pour une mantisse comportant nb bits, si on arrondit avec la fonction float() selon la technique dite "perfect rounding", on obtient $\varepsilon = \frac{1}{2^{nb}}$.

Si on utilise des mots en double précision, la précision machine est alors $\varepsilon = \frac{1}{2^{52}} \approx 2,22.10^{-16}$.

Pour déterminer directement la précision machine en utilisant python on peut utiliser **np.finfo** :

```
>>> np.finfo(np.float64).eps
2.2204460492503131e-16
>>> np.finfo(np.float32).eps
1.1920929e-07
```

Au final l'épsilon machine est ici de :

Simple précision	Double précision
$\varepsilon \approx 1,2.10^{-7}$	$\varepsilon \approx 2,2.10^{-16}$

5.4.2 Conditionnement

5.4.2.1 Définition

Pour A symétrique et inversible, on appelle conditionnement de A (que l'on note $cond(A)$), la valeur définit par :

$$cond(A) = \|A\|.\|A^{-1}\|$$

où $\|.\|$ est une norme sur l'espace des matrices.

Exemple :

$$\text{norme 1} \quad \|A\|_1 = \max_j \sum_i |a_{i,j}|$$

$$\text{norme infinie} \quad \|A\|_\infty = \max_i \sum_j |a_{i,j}|, \ldots$$

En général, il est très coûteux de calculer le conditionnement de A. En revanche, un certain nombre de techniques peuvent permettre de l'estimer.

On va montrer dans la suite qu'avec un conditionnement grand la résolution du système conduit à des problèmes de précision (cumul d'arrondis numériques).

5.4.2.2 Initialisation

Un premier calcul utilisant l'algorithme de Gauss défini précédemment est conduit en utilisant des nombres simple précision.

On crée une matrice A, dont on fixe la précision avec float32, puis on construit B de manière à être sûr que la solution $x = [2,-2]$ soit la solution du problème :

```
A=np.array([[0.2161, 0.1441],[1.2969001, 0.8648]],dtype=np.float32)
B=np.array([[ 2*A[0,0]-2*A[0,1]],[ 2*A[1,0]-2*A[1,1]]],dtype=np.float32)
```

On obtient alors $A' = \begin{bmatrix} 0,21610001 & 0,1441 \\ 1,29690015 & 0,86479998 \end{bmatrix}$ et $B' = \begin{bmatrix} 0,14400002 \\ 0,86420035 \end{bmatrix}$.

5.4.2.3 Triangularisation

```
for k in range(N):
    # Mise a zeros
    for i in range(k+1,M):
        #On modifie le second membre avant A
        B[i] -= B[k]*A[i,k]/A[k,k]
        A[i,:] -= A[k,:]*A[i,k]/A[k,k]
```

On obtient alors :

$$A = \begin{bmatrix} 2,16100007e-01 & 1,44099995e-01 \\ -1,19209290e-07 & -1,19209290e-07 \end{bmatrix}$$

$$B = \begin{bmatrix} 1,44000024e-01 \\ 2,38418579e-07 \end{bmatrix}$$

5.4.2.4 Remontée

Pour une précision de 32 bits on obtient alors $sol = \begin{bmatrix} 5,88888787 \\ -1,94309494 \end{bmatrix}$. Cette solution est très loin de la solution exacte.

Lorsqu'on réalise le calcul avec une précision de 64bits la solution en revanche est bien $sol = \begin{bmatrix} 2. \\ -2. \end{bmatrix}$.

```
for k in range(N-1,0,-1):   #-1 pour pas negatif
        for i in range(k):
            B[i] -= B[k]*A[i,k]/A[k,k]
            A[i,:] -= A[k,:]*A[i,k]/A[k,k]
sol= np.zeros((1,N))
for k in range(N):
    sol[0,k]=B[k]/A[k,k]
```

5.4.2.5 Calcul du résidu

On appelle résidu ou erreur résiduel r tel que :

$$r = \mathrm{B} - \mathrm{A}.\mathrm{X}$$

```
res=b-np.dot(A,X)
```

Ce résidu devrait normalement donner une indication sur la qualité de la solution.

Cependant, on obtient :

- Résidu pour une précision de 32 : $\begin{bmatrix} 0,00000000e+00 \\ 7,02010139e-07 \end{bmatrix}$
- Résidu pour une précision de 64 : $\begin{bmatrix} 0. \\ 0. \end{bmatrix}$

Alors que la solution est très différente, on obtient bien dans les deux cas un résultat proche de 0. Si on tient compte de la précision machine correspondant au calcul, pour le calcul en 32 bits on a en effet $\varepsilon \approx 1,2.10^{-7}$.

5.4.2.6 Calcul du conditionnement de A

On utilise la norme infinie pour déterminer la norme de A :

$$\|\mathrm{A}\|_\infty = \max_{1\leq i\leq n} \sum_{j=1}^{n} |a_{ij}| = \max(2.1617, 0.3602) = 2.1617$$

Exceptionnellement pour cet exemple on inverse la matrice A :

$$\mathrm{A}^{-1} = \begin{bmatrix} 14410000.00 & -86480000.00 \\ -21610000.00 & 129690000.0 \end{bmatrix} \Rightarrow \|\mathrm{A}^{-1}\|_\infty = 151300000$$

D'où le conditionnement de A :

$$cond(\mathrm{A}) = \|\mathrm{A}\|.\|\mathrm{A}^{-1}\| = 3,2.10^{8}$$

Le conditionnement de A est grand $cond(\mathrm{A}) \gg 1$.

Le résidu (dans le cas du calcul en 32bits) est petit puisque $\mathrm{B} - \mathrm{A}.\mathrm{X} = [0.00000000e+00; 7.02010139e-07]^{\mathrm{T}}$ alors que la solution exacte $[2-2]^{\mathrm{T}}$ est très différente de $\mathrm{X} = [5.88888787; -1.94309494]^{\mathrm{T}}$. Il apparaît donc que le résidu ne soit pas un indicateur totalement fiable.

D'autre part il semble que le conditionnement du système soit très fortement lié à la qualité de la solution (en terme d'influence des arrondis numériques) que l'on puisse espérer avoir. On va démontrer ceci dans la suite.

On voit qu'avec une précision de 64 bits on n'a pas ici de problème de résolution. En augmentant la précision sur les nombres (choix d'une précision double ou simple), on évite les problèmes liés aux erreurs d'arrondis.

Cependant on augmente la taille mémoire demandée pour stocker les données.

5.4.2.7 Système perturbé

Pour illustrer l'influence du conditionnement, on est amené à considérer un système perturbé (par des perturbations généralement supposées petites) $\tilde{A}.\tilde{X} = \tilde{B}$.

Ici on choisit de modifier A très légèrement et de rester dans la suite du calcul avec une précision de 64 bits sur les nombres.

On introduit sur un seul terme une perturbation de 1.10^{-7} correspondant donc de l'ordre de grandeur de l'epsilon machine quand on fait le calcul avec une précision de 32 bits et donc au type d'erreur introduite lors d'un calcul en simple précision.

```
A=np.array([[0.2161001, 0.1441],[1.2969001, 0.8648]],dtype=np.float64)
B=np.array([[ 0.144] ,[0.8642002]],dtype=np.float64)
```

La solution est alors $[-0,78653134; 2,17883068]$ et le résidu de $[0,0; 0,0]$.

On constate que pour une valeur de conditionnement élevé (loin de 1) le résultat (loin de la solution $[-2,2]$) est très sensible aux perturbations et par conséquent, aux approximations numériques.

De plus, le calcul du résidu n'est donc pas fiable pour un problème mal conditionné.

5.4.2.8 Influence des perturbations

Considérons le système $\tilde{A}.\tilde{X} = \tilde{B}$. Si $\tilde{B} = B + \delta B$ avec $||\delta B|| \ll ||B||$. A-t-on $||\delta X|| \ll ||X||$?

Calculons simplement, en utilisant l'inégalité de Milkowski :

$$A.X = B \Rightarrow ||A.X|| = B|| \quad \Rightarrow \quad ||A||.||X|| \geq ||B|| \quad \Rightarrow \quad \frac{1}{||X||} \leq \frac{||A||}{||B||}$$

Dans le cas du système perturbé au niveau de B :

$$A.(X + \delta X) = B + \delta B \Rightarrow A.\delta X = \delta B \Rightarrow \delta X = A^{-1}\delta B \quad \Rightarrow \quad ||\delta X|| \leq ||A^{-1}||.||\delta X||$$

par conséquent :

$$\frac{||\delta X||}{||X||} \leq cond(A)\frac{||\delta B||}{||B||}$$

Dans le cas du système perturbé au niveau de A :
Si $\tilde{A} = A + \delta A$ et $\tilde{X} = X + \delta X$ alors

$$(A + \delta A).(X + \delta X) = B \Rightarrow A.\delta X + \delta A.(X + \delta X) = 0 \Rightarrow \delta X = -A^{-1}.\delta A.(X + \delta X)$$

$$\Rightarrow \quad ||\delta X|| \leq ||A^{-1}||.||\delta A||.||X + \delta X||$$

soit au final :

$$\frac{||\delta x||}{||x + \delta x||} \leq cond(A).\frac{||\delta A||}{||A||}$$

Même si la perturbation est faible, que ce soit sur A ou B, si le conditionnement est grand, alors l'influence sur le résultat peut être important.

5.4.3 Choix du pivot

5.4.3.1 Problème de stabilité

Au-delà du problème de conditionnement l'algorithme de Gauss peut présenter des problèmes de stabilité.

Le pivot de Gauss est le terme $a_{i,i}$ de la matrice A au début de l'étape i. Étant donné que les calculs se font avec une arithmétique en précision finie, la grandeur relative du pivot influence la précision des résultats.

On peut rencontrer des pivots très petits sans que le problème soit mal conditionné :

$$\mathrm{A} = \begin{bmatrix} \varepsilon & 1 \\ 1 & 0 \end{bmatrix} \quad \text{et} \quad cond(\mathrm{A}) = 1$$

5.4.3.2 Exemple d'instabilité de l'algorithme de Gauss

Soit le système $\begin{cases} 0,0001.x_1 + 1,0.x_2 = 1 & (1) \\ 1,0000.x_1 + 1,0.x_2 = 2 & (2) \end{cases}$ pour lequel la solution exacte est $x_1 = 1,0001$ et $x_2 = 0,9998$.

Appliquons l'élimination de Gauss avec une arithmétique en base 10 et $nb = 3$.

On a $a_{1,1} = 0,0001$ le pivot et $m_{G1,2} = \dfrac{a_{2,1}}{a_{1,1}} = \dfrac{1}{0,0001} = 10000$ le multiplicateur de Gauss pour la deuxième ligne.
L'équation (2) devient alors :

$$\begin{aligned} (2) : \quad & 1.x_1 + 1.x_2 = 2 \\ (1).m_{G1,2} : \quad & -1.x_1 + -10000.x_2 = -10000 \\ (2) + (1).m_{G1,2} : \quad & -999.x_2 = -9998 \end{aligned}$$

Avec 3 digits significatifs on obtient alors $x_2 = 1$. On remplace x_2 dans (1) et on obtient $x_1 = 0$. C'est très loin de la solution attendue.

Or ceci ne se produit pas si on permute les lignes.

Ainsi, l'élimination de Gauss est un algorithme numériquement instable. Il est donc nécessaire d'introduire dans l'algorithme un mécanisme de permutation des lignes et/ou des colonnes afin d'éviter des petits pivots.

En conclusion, pour obtenir un résultat valable à l'issu d'une résolution de système linéaire il faut avoir un problème bien conditionné afin d'avoir un résultat précis, mais il faut aussi veiller à ce que l'algorithme soit numériquement stable afin de produire la bonne solution.

5.5 Technique de stockage et coût

5.5.1 Notion de coût

Le coût de résolution d'un problème fait naturellement intervenir le temps mis pour effectuer l'ensemble des opérations, mais aussi la taille mémoire requise.

Le temps peut être mesuré de deux façons :

- temps horloge : mesure globale du temps extérieur écoulé pendant la résolution, dépendante de la machine utilisée, du problème traité et de la compilation du programme. Le résultat est alors une estimation

Nombre de nœuds	5	13	49	149	449	999
Taille matrice	7.7	23.23	95.95	295.295	895.895	1995.1995
Taille mémoire	392 o	4ko	72 ko	696 ko	6.4 Mo	31.8 Mo

Table 5.1 – Evolution de la taille mémoire pour l'exemple du treillis Fig 5.1.

du temps pour effectuer un programme dans sa globalité, il n'est pas uniquement lié à l'algorithme employé.

- temps CPU (Central Processing Unit) : cette fois lié aux opérations effectuées.

Pour avoir un indicateur sur l'algorithme uniquement on utilise le nombre d'opérations en virgule flottante nécessaire à la résolution.

5.5.2 Coût du stockage et optimisation

5.5.2.1 Détermination de l'espace mémoire nécessaire pour une matrice pleine

La structure de données naïve utilisée pour stocker une matrice $M_{m,n}$ est un tableau à deux dimensions. Pour une matrice $M_{m,n}$ il faut au moins $m.n$ espaces mémoire (un pour chaque terme) de taille fixe pour représenter la matrice. Chaque terme est représenté classiquement par un flottant (double précision) codé sur 64 bits. Lors de simulations numériques il est courant d'avoir des applications avec des tailles de système de $n = 10^3$.

5.5.2.2 Réorganisation des équations du système

En simulations numériques, il est fréquent que chacune des équations à résoudre ne concerne qu'une faible partie de l'ensemble des inconnues du système. Par exemple, la ligne i peut être fonction de x_i et des ses deux plus proches voisins (x_{i-1} et x_{i+1}). Chaque ligne présente alors un grand nombre de zéros. On parle alors de matrices creuses.

Avec quelques opérations sur les lignes et les colonnes, il est souvent possible de réarranger le problème initial pour obtenir une matrice sous forme plus classique (zeros, ones, diagonale, bande, ligne de ciel,...).

5.5.2.3 Stockage des matrices creuses

L'idée est ici de ne stocker que les termes non nuls de la matrice. Il existe de nombreux formats pour cela. On va utiliser le format *Yale Sparse Matrix* qui stocke une matrice M de taille $m.n$ sous la forme de trois tableaux unidimensionnels. Soit k le nombre de termes non nuls de M :

- le premier tableau noté A est de longueur k. Il contient toutes les valeurs des entrées non nulles de M de gauche à droite et de haut en bas.
- le deuxième tableau est noté IA de longueur $m+1$ (le nombre de lignes plus un). L'élément i de IA contient l'index dans le tableau A de la première entrée non nulle de la ligne i de la matrice M. La ligne i de la matrice originale est composée des éléments de A depuis l'index IA(i) jusqu'à l'index IA($i+1$) – 1.
- le troisième tableau, noté JA, de longueur k, contient le numéro de la colonne de chaque élément de A

Les tableaux IA et JA ne stockent que des nombres entiers.

5.5.2.4 Stockage des matrices bande

On parle de matrice Bande lorsque les cœfficients non nuls d'une matrice creuse se regroupent autour de la diagonale. Tous les cœfficients $a_{i,j}$ non nuls sont situés dans une bande délimitée par des parallèles à la diagonale principale.

Une telle bande est déterminée par deux entiers k_1 et k_2 positifs ou nuls, tels que : $a_{i,j} = 0$ si $j < i - k_1$ ou $j > i + k_2$

Toutes les diagonales situées entre les deux diagonales contenant les éléments non nuls les plus éloignés de la diagonale principale sont stockées comme colonnes d'une matrice rectangulaire. Le nombre de lignes de cette matrice est égal à l'ordre de la matrice creuse. Son nombre de colonnes est égale à la largeur de la bande.

5.5.3 Coût en opération et optimisation

Le fait de stocker un nombre important de termes ne pose pas uniquement un problème pour le stockage.

En effet, le nombre d'opérations pour résoudre le système A.X = B est lié aussi au nombre de termes stockés de la matrice.

5.5.3.1 Complexité en triangularisation

On s'intéresse ici à la complexité de la triangularisation pour une matrice $n.n$ sans tenir compte de la recherche du pivot.

Considérons l'étape $i \in [\![1, n]\!]$. Pour chaque ligne k allant de $i+1$ à n, on recense les opérations suivantes :

- une division (div) afin de calculer, pour toute la ligne le cœfficient permettant d'obtenir des zéros sous le pivot ($m_{Gi,k}$)
- une addition (add) et une multiplication (mult) permettant de mettre à jour chaque cœfficient de la ligne k, ce qui correspond à toutes les colonnes de numéros $j \in [\![i, n]\!]$
- une addition (add) et une multiplication (mult) permettant de mettre à jour le second membre de la ligne k, i.e. b_k.

Le nombre d'opérations (nop) avec prise en compte du second membre s'écrit donc

$$\begin{aligned}
nop &= \sum_{i=1}^{n-1}\sum_{k=i+1}^{n}\left[1.div + \left(\sum_{i=j}^{n} 1.add + 1.mult\right) + 1.add + 1.mult\right] \\
&= \sum_{i=1}^{n-1}\sum_{k=i+1}^{n}\left[1.div + (n-i+1).add + (n-i+1).mult\right] \\
&= \sum_{i=1}^{n-1}(n-i).div + (n-i).(n-i+1).add + (n-i).(n-i+1).mult \\
&= \sum_{i=1}^{n-1}(n-i).div + \sum_{i=1}^{n-1}(n-i).add + \sum_{i=1}^{n-1}(n-i)^2.add + \sum_{i=1}^{n-1}(n-i).mult + \sum_{i=1}^{n-1}(n-i)^2.mult \\
&= \frac{(n-1).n}{2}.div + \frac{(n-1).n}{2}.add + \frac{n.(n-1).(2n-1)}{6}.add + \frac{(n-1).n}{2}.mult + \frac{n.(n-1).(2n-1)}{6}.mult
\end{aligned}$$

5.5.3.2 Complexité en remontée

Considérons l'étape $i \in [\![1,n]\!]$. Le calcul de x_i par l'algorithme de remontée demande 1 division, $n-i$ additions et $n-i$ multiplications. De plus le calcul de x_n requiert 1 division. Ainsi :

$$nop = 1.div + \left(\sum_{i=1}^{n-1} (n-i).add + (n-i).mult + 1.div \right) = \frac{(n-1).n}{2}.add + \frac{(n-1).n}{2} mult + n.div$$

5.5.3.3 Complexité de l'ensemble

Opérations	Division	Additions	Multiplications
coût triangularisation	$\frac{n.(n-1)}{2}$	$\frac{n.(n-1).(n+1)}{3}$	$\frac{n.(n-1).(n+1)}{3}$
coût remontée	n	$\frac{n.(n-1)}{2}$	$\frac{n.(n-1)}{2}$
coût total	$\frac{n.(n+1)}{2}$	$\frac{n.(n-1).(2.n+5)}{6}$	$\frac{n.(n-1).(2.n+5)}{6}$

Au final la partie triangularisation est de l'ordre n^3 ($nop = \mathrm{O}(n^3)$), la remontée est de l'ordre n^2 donc au total l'algorithme est de l'ordre 3 $\mathrm{O}(n^3)$.

5.5.3.4 Complexité du pivotage

L'opération de pivotage nécessite lui aussi quelques opérations pour rechercher le meilleur pivot sur la ligne. A chaque ligne i, il faudra donc $n-i$ opérations élémentaires :

$$nop = \sum_{k=1}^{n-1} n-k = \sum_{k=1}^{n-1} k = \frac{n.(n-1)}{2} \text{ comparaisons}$$

Ceci ne modifie donc pas l'ordre de complexité de l'algorithme qui reste de 3.

5.5.4 Utilisation de librairies

Il est plus efficace d'utiliser des libraires que de reprogrammer soi-même des routines existantes.

D'un point de vu pratique, l'usage de librairie permet d'augmenter la lisibilité du code. De plus les librairies qui sont programmées de façon plus efficace sont souvent plus performantes.

Le sous-module **linalg** de **numpy** permet la résolution de systèmes linéaires.

En remplaçant toute la partie résolution par x=np.linalg.solve(A,b) on obtient les résultats suivants :

```
x=np.linalg.solve(A,b)
```

5.6 Application à un problème de treillis

La Fig 5.1 ci-contre représente un treillis de 5 nœuds et 7 barres. Chaque barre a une longueur de 6 m. La hauteur de l'ouvrage est de 3m.

Le nœud central est soumis à une charge de 15 kN (par exemple : effort exercé par un véhicule au centre du pont).

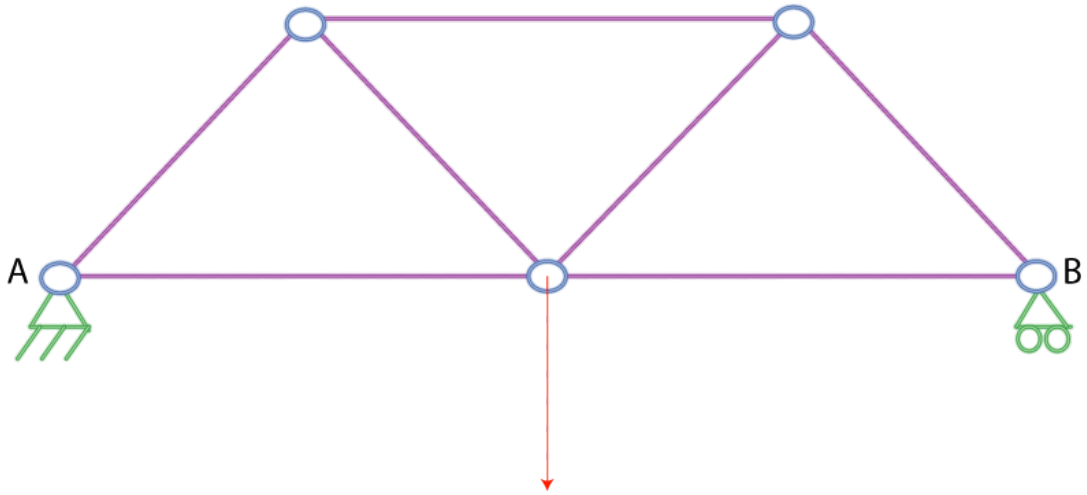


Figure 5.1 – Structure métallique treillis

Par une application du PFS à l'ensemble de la structure on peut déterminer les efforts dans chaque appui.

Soit $-F.\vec{y}$ l'effort central. Alors $\overrightarrow{F_{ext_A}} = \frac{F}{2}.\vec{y}$ et $\overrightarrow{F_{ext_B}} = \frac{F}{2}.\vec{y}$.

Les inconnues que l'on cherche à déterminer sont les forces de traction qui s'exercent sur chaque barre.

L'application de l'équilibre à chacun des nœuds permet d'obtenir alors le système d'équations à résoudre.